Ein generisches Monitoringsystem für akustische Datenströme

Diplomarbeit

Jan Henrik Wild

Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik III

01.03.2005

Remember, Information is not knowledge; Knowledge is not wisdom; Wisdom is not truth; Truth is not beauty; Beauty is not love; Love is not music; Music is the best.

Frank Zappa

Danksagung

Bei der Entstehung der vorliegenden Diplomarbeit standen mir einige Personen hilfreich zur Seite, denen ich im Folgenden danken möchte:

Zunächst möchte ich mich bei der Arbeitsgruppe Multimedia-Signalverarbeitung der Universität Bonn, und dabei insbesondere bei Professor Dr. Clausen für die Vergabe dieses Diplomarbeitsthemas bedanken. Weiterhin danke ich meinem Betreuer Dr. Frank Kurth für die konstruktive Zusammenarbeit und die positive Unterstützung während der gesamten Dauer der Arbeit. Weiterhin waren die Anregungen von Andreas Ribbrock und Rolf Bardeli eine Hilfe bei der Konzeptionierung des Systems.

Ich möchte meiner Familie und meinen Freunden danken, die mich in der gesamten Zeit vermutlich mehr ertragen als erlebt haben und mir in den richtigen Momenten durch Unterstützung oder Ablenkung geholfen haben.

Vor allem möchte ich mich bei Christopher Gies, Kai Starke, Julia von Selchow, Sebastian Wild und Kai Leder für Verbesserungsvorschläge und Feedback bedanken.

Mein besonderer Dank gilt Eleni Orfanidou, die immer für mich da war und mir Stärke gab.

Inhaltsverzeichnis

1 Einleitung und Übersicht 1
- 1.1 Gliederung der Diplomarbeit . . . 3
- 1.2 Hinweise zur Notation . . . 3

2 Audiomonitoring 4
- 2.1 Der Monitoringbegriff . . . 4
- 2.2 Inhaltsbasierte Audioidentifikation . . . 7
- 2.3 Der Audiofingerabdruck . . . 10
- 2.4 Ein generisches Audioidentifikationssystem . . . 12
  - 2.4.1 Front-End . . . 14
  - 2.4.2 Fingerabdrucks-Modellierer . . . 24
  - 2.4.3 Distanzmetrik . . . 24
  - 2.4.4 Suchmethoden . . . 24
  - 2.4.5 Hypothesen-Test . . . 25
  - 2.4.6 Datenbank-Pflege . . . 25
- 2.5 Aktuelle Audioidentifikationssysteme . . . 26
  - 2.5.1 AudioID . . . 26
  - 2.5.2 Shazam Ent. / Wang . . . 26
  - 2.5.3 AudioDNA . . . 27
  - 2.5.4 AudioHashing . . . 28
  - 2.5.5 Audentify! . . . 28
- 2.6 Audioklassifikation . . . 29
- 2.7 Audiomonitoringsysteme . . . 34

3 Multimedia-Frameworks 37
- 3.1 Frameworks . . . 38
- 3.2 Multimedia-Frameworks . . . 40
- 3.3 Aktuelle Multimedia-Frameworks . . . 43
  - 3.3.1 Quicktime . . . 43
  - 3.3.2 Java Media Framework . . . 44
  - 3.3.3 Linux . . . 45
  - 3.3.4 Helix / RealMedia . . . 46
  - 3.3.5 Windows Media . . . 46
  - 3.3.6 DirectX . . . 47
  - 3.3.7 MPEG-21 . . . 49

4 Entwurf des GenMAD-Systems 50
- 4.1 Verwandte Ansätze & Motivation . . . 51
  - 4.1.1 Wahl des Multimedia-Frameworks . . . 52
- 4.2 Das GenMAD-System . . . 53
  - 4.2.1 Grundlegendes Konzept des GenMAD-Systems . . . 53
  - 4.2.2 Zentrale Aspekte der GenMAD-Applikation . . . 56
  - 4.2.3 Filter . . . 58

5 Implementation 59
5.1 GenMAD . . . 59
5.1.1 Zentrale Klassen von GenMAD . . . 61
5.1.2 DirectShow-Events . . . 63
5.1.3 Hinzufügen neuer Filter . . . 63
5.1.4 Speichern & Laden . . . 64
5.1.5 Hinzufügen von Mediendateien als Datenquelle . . . 64
5.1.6 Filter-Konfiguration . . . 65
5.1.7 Zeit und Uhren . . . 65
5.1.8 Datenfluss, Abspielkontrolle & Seeking . . . 65
5.1.9 Visualisierung . . . 66
5.2 Das Filter-SDK . . . 67
5.2.1 Medienformat-Abgleich . . . 68
5.2.2 Datenübertragung . . . 68
5.2.3 Konfiguration . . . 69
5.2.4 Seeking . . . 70
5.2.5 Laden/Speichern . . . 70
5.3 Implementierte GenMAD-Filter . . . 70
5.3.1 MonitoringAudentifyFilter . . . 70
5.3.2 MonitoringEventNullRenderer . . . 70
5.3.3 MonitoringEventVisualizer . . . 71
5.3.4 MonitoringEventSplitter . . . 71
5.3.5 MonitoringEventMerger . . . 71
5.3.6 MonitoringEventExcluder . . . 72
5.3.7 MonitoringEventIncluder . . . 73
5.3.8 MonitoringXMLWriter . . . 73
5.3.9 MonitoringXMLReader . . . 73
5.3.10 MonitoringClassifier . . . 75
5.3.11 Weitere filterbasierte Problemlösungen . . . 76

6 Anwendungen, Tests und Bewertungen 77
6.1 Test des Klassifikationsfilters . . . 77
6.1.1 Datenauswahl . . . 77
6.1.2 Audiomerkmale . . . 78
6.1.3 Einfluss der Anzahl von Trainings- und Testdaten . . . 79
6.1.4 Einfluss verschiedener Abtastraten . . . 80
6.1.5 Einfluss der Fensterweite . . . 80
6.1.6 Vergleich verschiedener Nachbearbeitungsmechanismen . . . 80
6.1.7 Einfluss der Parameter des k-NN-Verfahrens . . . 81
6.1.8 Parallelklassifikation mit verschiedenen Abtastraten . . . 81
6.1.9 Klassifikationsleistung anhand von Radiomitschnitten . . . 82
6.2 Performanz des GenMAD-Systems . . . 84
6.2.1 Performanz der entwickelten Filter . . . 84
6.2.2 Performanz von GenMAD . . . 85
6.3 Anwendungen . . . 85

7 Zusammenfassung, Diskussion und Ausblick 88

Anhang 91
Anhang A: Hinweise zur Verwendung des Filter-SDK 91
Anhang B: Hinweise zur Verwendung von GenMAD 92
Anhang C: Benutzerhandbuch 93
Anhang D: Technische Dokumentation von GenMAD 97
Anhang E: Technische Dokumentation des Filter-SDK 114
Anhang F: Bedienung des Klassifikationsfilters 124
Anhang G: Inhalt der beiliegenden CD 125

Literaturverzeichnis 126

Kapitel 1

Einleitung und Übersicht

Mit steigenden Anwenderzahlen und immer schnelleren Internet-Zugängen, größeren Speicherkapazitäten und höher aufgelösten Daten stiegen Menge und Anteil von Multimedia-Inhalten in den letzten Jahren rasant an. Ob Internet-PC, Digitalkamera oder DVD-Player – für viele Menschen ist der Umgang mit multimedialen Inhalten alltäglich geworden. Virtuelle Ladentheken im Online-Musikgeschäft setzten im Jahr 2004 ca. 270 Mio. Dollar um und werden dies nach Schätzungen des Marktforschungsunternehmens Jupiter Research bis zum Jahr 2009 auf 1,7 Milliarden Dollar ausweiten. Dabei werden riesige Datenbanken verwaltet: Apples iTunes [5], mit 70% Marktanteil Branchenprimus, bietet Musik aus einem Bestand von über 800.000 Titeln an, Neueinsteiger Microsofts MSN Music [72] will dies mit einer Million Titeln noch übertrumpfen. Gleichzeitig zeichnet sich auch schon die nächste Plattform zur Verbreitung multimedialer Inhalte ab, nämlich in Form von Mobiltelefonen und anderen mobilen Techniken, in die bereits bestehende Technologien integriert werden, z.B. Videokonferenzen oder mobile Musikbiblio-theken.

Doch mit der steigenden Datenflut ergeben sich sowohl für Privatpersonen als auch für Firmen neue Herausforderungen im Umgang mit Multimediadaten: Wie sollen die Inhalte katalogisiert, verwaltet und wiederverwendet werden? Wie kann der Überblick über riesige Datenbestände behalten oder wie können neue Geschäftsbereiche darauf aufgebaut werden?
Im Umgang mit textorientierten Daten scheint dies einfacher zu sein: Zum einen benötigen diese deutlich weniger Speicherplatz, zum anderen existieren zahlreiche Ansätze aus dem Bereich des *Information Retrieval*, um durch effiziente Vergleichsverfahren oder Inhaltsanalysen Textsammlungen zu strukturieren und zu verwalten. Suchmaschinen bieten darauf aufbauend mehr oder weniger komfortable Suchmasken, um Inhalte in der Unstrukturiertheit des World Wide Web (WWW) zu finden.

Einer der Väter des WWW, Tim Berners-Lee, entwarf daher das Konzept des *Semantic Web* [119], welches durch Verwenden universeller, XML-basierter Regeln die semantische Repräsentation von in Daten verpacktem Wissen ermöglichen und dadurch die automatisierte Verarbeitung von Daten entscheidend verbessern und vereinfachen soll. In ähnlicher Weise stellt *MPEG-7* [70] einen Standard für die Beschreibung von Multimediainhalten durch Deskriptoren dar. Dieser könnte z.B. von Suchmaschinen, ob Desktop- oder webbasiert, angewandt werden, um das Auffinden von Inhalten nach bestimmten Kriterien zu erlauben.

Das tatsächliche maschinelle *Verstehen* von Multimediainhalten, *Semantic Computing*, erfordert aber noch mehr als Deskriptoren. Erst ein tieferes Verständnis von Inhalten, welches sich nicht beigefügten und eingebetteten Metadaten bedient, sondern den Inhalt selbst verarbeitet, erlaubt z.B. automatisiertes Bewerten und Katalogisieren von Multimediadaten.

In diesem Kontext fällt dem *Multimedia Information Retrieval* eine besondere Rolle zu. Dieses junge Teilgebiet der Informatik befasst sich mit verschiedenen Ansätzen, um multimediale Daten inhaltsbasiert analytisch zu verarbeiten und ist somit ein wichtiger Puzzlestein des Semantic Computing. Ein wichtiger Teilbereich, der in dieser Diplomarbeit behandelt wird, ist das *Audio*

Information Retrieval, welches der Gewinnung von semantischen Beschreibungen aus beliebigen Audiodaten dient.

Hier entwickelte Technologien werden z.B. von Mobilfunkanbietern wie Vodafone oder O2 dazu verwendet, in *Query-By-Mobile* genannten Verfahren Musik zu identifizieren, bei dem ein Benutzer sein Mobiltelefon z.B. vor ein Radio hält, in dem ein ihm unbekanntes Lied läuft. Da in solchen Verfahren eingehende Audiosignale üblicherweise mit Hilfe einer Menge von Musikstücken verglichen werden, die in einer Datenbank repräsentiert sind, spielt deren Umfang der darin gespeicherten Titel eine entscheidende Rolle. Philips beispielsweise hat sich daher für ihren *AudioID*-Service die Dienste von Gracenote, vormals *CDDB*, gesichert, in deren Datenbank Metadaten für vier Millionen Titel abgelegt sind [85].

Andere Firmen wie MusicMatch [75] integrieren eine solche – allgemein *Audio Fingerprinting* genannte – Identifikationstechnologie in Software zur automatischen Benennung von Musikstücken oder bieten neuartige Dienste an, die ein Musikstück *klassifizieren* und entsprechende Vorschläge zu ähnlichen Stücken machen.

Ein dazu eng verwandtes Gebiet, das sich aus der Notwendigkeit von Kontrollmechanismen ergibt, ist das sogenannte *Audiomonitoring*. Dieses befasst sich mit der Echtzeitüberwachung von Audioströmen und verwendet Algorithmen aus der Audioidentifikation und -klassifikation. Anwendung ergibt sich hierbei z.B. aus Musik-Tauschbörsen, bei denen Musikstücke illegal ausgetauscht werden. Da deren Benutzer wie im Fall von Napster [77] einfache namensbasierte Filtermechanismen umgehen konnten, hat die Musikindustrie ein starkes Interesse an inhaltsbasierten Kontrollmechanismen [90]. Ebenso könnten Benutzer solche Identifikationsfunktionalitäten nutzen, um Inhalte zu verifizieren [109]. Verwandte Szenarien ergeben sich aus der Protokollierung von Radio- und Fernsehsendungen, um Lizenzgebühren abzurechnen oder das Senden von Werbeblöcken zu überprüfen. Die in diesem Zusammenhang oft genannten Verfahren zur Verwendung von sogenannten *Audiowasserzeichen*, also mittels Methoden der Psychoakustik eingebetteten Metadaten zur Markierung eines Stückes, die nur minimale Veränderung des Klanges hervorrufen, sind ebenfalls nur in bestimmten Fällen anwendbar und können im Allgemeinen ein inhaltsbasiertes Erkennen nicht ersetzen.

Die Szenarien, in denen Audiomonitoring angewandt werden kann, sind jedoch keineswegs auf Musik beschränkt, obwohl es sich z.Zt. wahrscheinlich um das wirtschaftlich umfangreichste Anwendungsszenario handelt. Vielmehr werden potentiell alle Arbeitsfelder umfasst, in denen akustische Daten verarbeitet werden, wie z.B. im Falle von Projekten zum automatisierten Monitoring von Tierstimmen, die momentan von der Arbeitsgruppe Multimedia-Signalverarbeitung der Universität Bonn in Kooperation mit verschiedenen Partnern aus der Biologie und dem Bundesamt für Naturschutz in Angriff genommen werden.

Obwohl – wie in Kapitel 2 noch genannt – vielfältige Ansätze existieren, um Audiosignale zu identifizieren, zu klassifizieren und zu überwachen, sind diese jedoch in nur sehr eingeschränktem Maße austausch- und vergleichbar. Darüber hinaus fehlt oftmals ebenso eine Möglichkeit zur flexiblen Verwendung verschiedener Datentypen wie eine Schnittstelle zur Einbindung von fremden Algorithmen.

Ziel der vorliegenden Diplomarbeit ist daher die Konzeptionierung und Realisierung eines generischen Monitoringsystems für akustische Datenströme. *Generisch* bezeichnet im vorliegenden Fall die Eigenschaft, beliebige Monitoring-Algorithmen dynamisch über eine definierte Schnittstelle verwenden zu können. Dies schließt auch Möglichkeiten zur Filterung und Visualisierung derjenigen Daten mit ein, die die eingebundenen Algorithmen erzeugen. Auch wenn dies nicht die Definition eines Standards zur Arbeit mit Audiomonitoring-Verfahren darstellt, so soll das System doch auf die Notwendigkeit eines solchen Systems hinweisen und praxisnah Vor- und Nachteile verdeutlichen. Die Erweiterbarkeit des Systems ist daher wesentlicher Bestandteil des zugrunde liegenden Konzeptes.

1.1 Gliederung der Diplomarbeit

Die vorliegende Diplomarbeit ist folgendermaßen aufgebaut: Im nachfolgenden Kapitel 2 werden allgemeine Konzepte und bestehende Ansätze zum Gebiet des Audiomonitorings und dessen Anwendungen vorgestellt. Das Kapitel 3 beschäftigt sich mit technischen Multimedia-Frameworks, welche die maßgebliche Grundlage des in dieser Arbeit entwickelten generischen Monitoringsystems darstellen. Das in Kapitel 4 erläuterte neuartige Konzept dieses Monitoringsystems stellt einen wesentlichen Bestandteil der im Rahmen dieser Diplomarbeit geleisteten wissenschaftlichen Arbeit dar. Die Implementierung des Systems wird in Kapitel 5 erläutert und im Kapitel 6 verschiedenen Tests und Anwendungen unterzogen. Das abschließende Kapitel 7 fasst die Ergebnisse dieser Diplomarbeit zusammen und soll mittels einer Diskussion einen Ausblick auf mögliche Erweiterungen und fortführende Maßnahmen bieten.

Der Anhang schließlich umfasst sowohl das Benutzerhandbuch als auch die technische Dokumentation des entwickelten generischen Monitoringsystems.

Sämtliche im Rahmen dieser Diplomarbeit entwickelten Komponenten liegen als kompilierte Versionen und im Quelltext auf CD bei.

1.2 Hinweise zur Notation

Neu eingeführte Begriffe sind ebenso wie mathematische Zeichen und betonte Textelemente durch kursive Schriftweise hervorgehoben, z.B. $f(x)$. Technische Bezeichnungen wie Klassen-, Schnittstellen- oder Funktionsnamen sind wie folgt durch einen besonderen Schrifttyp gekennzeichnet: `IMonitoringMaster`. In dieser Arbeit verwendete Quellcode-Passagen sind ebenfalls mit dieser Schrift versehen, sind jedoch zusätzlich grau unterlegt, um sie vom restlichen Textfluss abzuheben: `for (int i=0; i<n; i++)`

Wenn es nötig war, sich in einem Satz einer Personifizierung zu bedienen, so wurde stets die maskuline Form gewählt, z.B. *der Benutzer*. Dies geschah keineswegs, um weibliche Leser zu irritieren, sondern vielmehr, um ein Textbild zu bewahren, dass dem gewohnten Lesefluss möglichst dienlich sein sollte.

Kapitel 2

Audiomonitoring

In diesem Kapitel werden der Begriff des Audiomonitorings, verwandte Bereiche und darauf basierende Anwendungen erläutert. Dazu wird zunächst im Abschnitt 2.1 der Begriff des (Multimedia-)Monitorings definiert und der Zusammenhang der einzelnen Teilkomponenten im allgemeinen Fall besprochen. In den beiden darauf folgenden Unterkapiteln wird dies konkretisiert, indem eine zentrale Komponente des Audiomonitorings, die inhaltsbasierte Audioidentifikation und der damit verbundene Audiofingerabdruck erläutert werden, bevor im Abschnitt 2.4 ein generisches Audioidentifikationssystem beschrieben wird. Beispiele existierender Systeme und der ihnen jeweils zugrunde liegenden Theorien sind im Kapitel 2.5 aufgeführt. Der Abschnitt 2.6 befasst sich mit dem eng verwandten Thema der Audioklassifikation und zeigt sowohl Technologien als auch Implementationen. Das abschließende Unterkapitel 2.7 komplettiert das in diesem zweiten Kapitel besprochene Thema des Audiomonitorings und gibt eine Übersicht über Audiomonitoringsysteme.

2.1 Der Monitoringbegriff

Monitoring stellt neben Such- und Synchronisationsaufgaben eine dritte zentrale Komponente des Multimediaretrievals dar und bezeichnet das systematische Überwachen von Datenströmen auf bestimmte zu definierende Ereignisse hin (engl. „to monitor“ – abhören, überwachen, kontrollieren).

Monitoring findet zahlreiche Anwendungen, etwa in automatischer Inhaltsanalyse von Videosignalen, z.B. zum Auffinden von Firmenlogos [19], der Überwachung des Straßenverkehrs anhand des Geräuschpegels [80] oder der automatischen Transkription von Nachrichtenbeiträgen eines Fernseh- oder Radioprogramms. Ein weiteres Anwendungsgebiet ist die akustische Kontrolle von Grenzwertüberschreitungen bei Maschinen, wie etwa Laufgeräuschen eines Motors.

Entscheidender Unterschied zwischen Monitoring und Suche ist die Echtzeit-Anforderung, d.h. die zu untersuchenden Datenströme liegen erst zur Bearbeitungszeit vor.

Wir betrachten hier diskrete Datenströme – auch abstrakt als *Dokumente* bezeichnet – der Form $x \subseteq \mathbb{Z} \times X$, die von Monitoringsystemen über die Zeit, diskret durch $\mathbb{Z}$, im endlichen Wertebereich X bearbeitet werden [57]. Kleinste Elemente eines solchen Datenstromes notieren wir mit x_i. Im folgenden sei $t : \mathbb{Z} \times X \rightarrow \mathbb{Z}$ die Projektion auf den Zeitpunkt eines solchen Datenelementes.

Im weiteren Verlauf der vorliegenden Arbeit relevant ist weiterhin die folgende, aus der Signalverarbeitung bekannte Kausalitätsanforderung [57]: Das Monitoringsystem kann zu einem Zeitpunkt $\tau \in \mathbb{Z}$ nur auf solchen Eingabedatenelementen x_i arbeiten, für die gilt: $t(x_i) \leq \tau$.

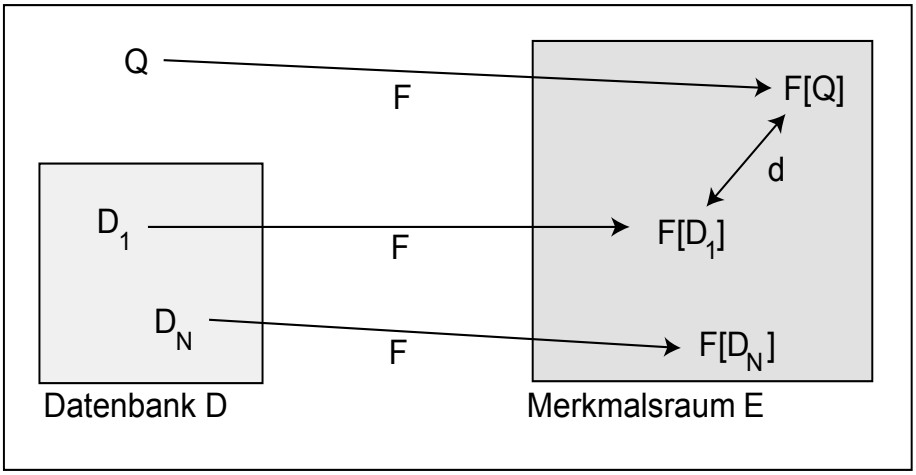


Abbildung 2.1: Vergleich von Anfrage und Datenbank

In der Praxis verfügt das Monitoringsystem nur über Daten aus einem Zeitfenster der Länge T, d.h. die zu bearbeitende Menge A der Eingangsdaten eines Monitoringsystems zu einem Zeitpunkt $\tau \in \mathbb{Z}$ ist definiert durch

$$A = \{ x_i \mid \tau \geq t(x_i) \geq \tau - T \}.$$

Gesucht wird beim Monitoring diejenige Menge von Dokumenten einer vorliegenden Multimedia-Datenbank, die an einer oder mehreren Stellen mit der Anfrage (engl. *Query*) – oder je nach Verfahren mit Teilen davon – in definiertem Maße perzeptuell oder nach sonstigen Kriterien übereinstimmt. Für den Vergleich werden sowohl Query wie abgefragte Datenbank-Informationen in einen Merkmalsraum E überführt und mittels eines Ähnlichkeitsmaßes (engl. *retrieval status value*, *RSV*) untersucht [27]. E muss dabei stark diskriminierend und somit hochgradig problemspezifisch sein.

Formal ausgedrückt:
Gegeben sei eine Dokumenten-Datenbank $D = (D_1, .., D_N)$ von Dokumenten $D_i \subseteq M$, wobei M eine Menge ist, z.B. $M = \mathbb{Z} \times X$. Zu einer Anfrage $Q \subseteq M$ wird diejenige Treffermenge $T_D^{d,\varepsilon}$ gesucht, die sich mittels einer perzeptuellen Vergleichsoperation $d : P(M') \times P(M') \rightarrow \mathbb{R}_{\geq 0}$ unter Verwendung einer Transformation $F : M \rightarrow M'$ aus D ergibt:

$$T_D^{d,\varepsilon}(Q) = \{ i \in [1..N] \mid d(F[Q], F[D_i]) \leq \varepsilon \}.$$

Dabei ist *P(M')* die Potenzmenge von *M'*. Es sei angemerkt, dass je nachdem, ob es sich bei der Vergleichsoperation *d* um ein Fehler- oder um ein Distanzmaß handelt, statt der notierten Ähnlichkeitsbedingung ≤ auch ein Unähnlichkeitskriterium ≥ verwendet werden kann.

Neben der Suche nach einer Treffermenge ergibt sich im Bereich des Monitorings zusätzlich die Anforderung, zu einem festen Q alle Stellen aller Dokumente D_i zu finden, die entsprechend oben genanntem Schema mit der Anfrage übereinstimmen, etwa zum Auffinden von Werbejingles in einem Radiostream. Die Grenzen zwischen Suche und Monitoring können aber je nach Einsatzgebiet und Zweck auch weniger klar sein.

Je nach Wahl der Vergleichsoperation handelt es sich bei der Suche um eine Identifikation, eine Klassifikation oder um eine Ähnlichkeitsbewertung: Im Fall der Identifikation wird das (u.U. mehrfache) Vorhandensein der Anfrage in der Datenbank überprüft, während die Klassifikation

die Anfrage anhand eines Ähnlichkeitsmaßes in eine bestimmte Klasse einstuft. Dies kann auch ohne Verwendung einer Datenbank geschehen. Möglich ist auch die Verwendung einer Ähnlichkeitsbewertung, wie z.B. eines Rankingverfahrens. In diesem Fall werden den Trefferkandidaten quantitative Bewertungen zugeordnet, die sich aus ihrer Ähnlichkeit mit dem gespeicherten Dokumentenbestand ergeben. Aufgrund der vielfältigen möglichen Darstellungen der Signale gibt es eine Vielzahl von Ähnlichkeitsmaßen, die sich meist der Wahrscheinlichkeitstheorie und der Statistik bedienen, um ein Entscheidungsschema bereitzustellen [27].

Die Güte eines Monitoringsystems ergibt sich aus zweierlei Qualitätskriterien für Information-Retrieval-Verfahren [37]: Zum einen durch die technisch gut messbare Effizienz, d.h. den möglichst sparsamen Umgang mit Ressourcen wie Rechenzeit und Speicherplatz, zum anderen durch die Effektivität, d.h. „die Fähigkeit des Systems, den Nutzenden die benötigte Information bei möglichst geringen Kosten an Zeit und Anstrengung anzubieten“ [37]. Genauer untersucht man die Relevanz zwischen Anfrage und Datenbestand, gegeben durch die Größen *Precision* (Anteil der relevanten Dokumente unter den gefundenen Dokumenten) und *Recall* (Anteil der relevanten Dokumente, die gefunden wurden). Natürlich muss für eine solche Analyse der zurückgelieferten Treffer die tatsächlich mögliche Treffermenge bekannt sein.

Optimal ist eine Treffermenge, wenn beide Größen einen Wert von 1 haben, wenn nämlich genau alle relevanten Dokumente gefunden wurden. Um das im Allgemeinen bestehende Problem des Abwägens zwischen Precision p und Recall r zu lösen, entwickelte Van Rijsbergen [111] ein Einheitsmaß e mit Gewichtung β,

$$e_\beta(p,r) := \frac{(\beta^2+1)\cdot p\cdot r}{\beta^2\cdot p\cdot r}.$$

In der Praxis kann die Aussagekraft der Relevanz in Abhängigkeit des tatsächlichen Inhaltes der vorliegenden Datenbank aber begrenzt sein.

Weitere Herausforderungen des Monitorings ergeben sich vor allem aus folgenden Bereichen:

- **Mögliche Unstrukturiertheit der Anfrage**: Im Falle von „intuitiven“ Eingabemasken muss die Eingabe des Benutzers für die weitere Verarbeitung formalisiert werden. Dies betrifft z.B. Verfahren, bei denen die Ähnlichkeit des Datenbestandes zu einer gepfiffenen Melodie überprüft werden soll.

- **Verzerrte oder verrauschte Eingangsdaten**: Die zu verwendenden Signaldaten können durch Hintergrund- oder Störgeräusche beeinflusst sein oder durch den Einsatz minderwertiger Hardware wie Mikrofone oder Wandler eine geringe Signalgüte aufweisen.

- **Sehr große und/oder unstrukturierte oder heterogene Datenbanken**: Dazu zählen Datenbanken mit variablen Dokumententypen und dezentrale Datenbanken, deren Synchronisierung die Echtzeit-Anforderung erschwert, ebenso wie riesige Dokumente teils komplexer Datentypen (Kombination verschiedener Typen, Objekt-Referenzen, Einbettung von Objekten), sowie die Aktualität der Daten.

- **Subjektive Semantik**: Das subjektives Ähnlichkeitsempfinden des Benutzers kann im Gegensatz zum definierten Ähnlichkeitsmaß des Monitoringsystems stehen.

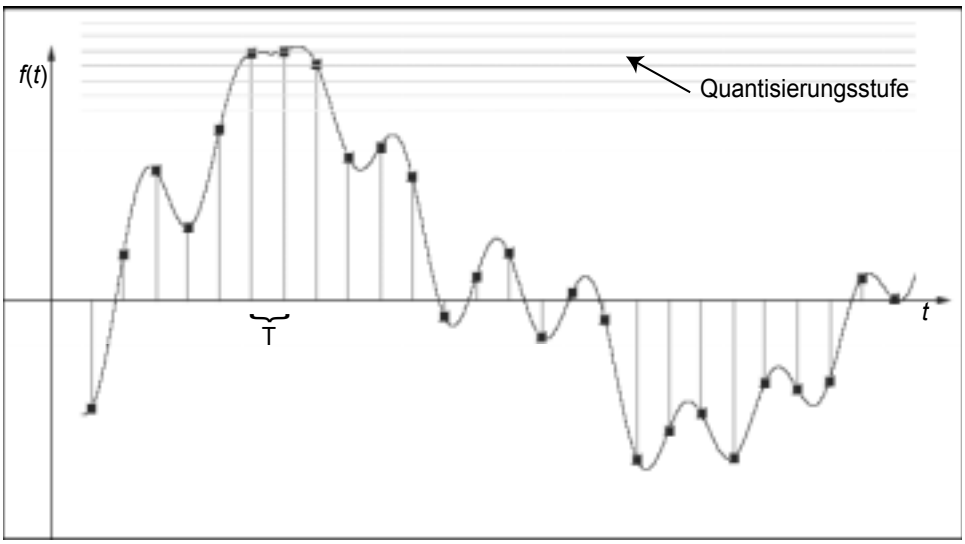


Abbildung 2.2: Digitalisierung eines kontinuierlichen Signals mittels Abtastung und Quantisierung

2.2 Inhaltsbasierte Audioidentifikation

Um das Prinzip des Monitorings auch auf akustische Datenströmen anwenden zu können, bedarf es zunächst einiger grundlegenden Definitionen und Begriffe aus den Bereichen Akustik und Digitale Signalverarbeitung.

Akustische Geräusche entstehen durch Dichtemodulation in einem elastischen Medium – im vorliegenden Kontext meist der Luft. Als Folge dieser lokalen Verdichtungen entstehen *Schallwellen*, kontinuierliche mechanische Schwingungen. Treffen Schallwellen auf eine Membran, so wird diese ebenfalls zum Schwingen angeregt. Koppelt man nun eine solche Membran mit einer Drahtspule, so lässt sich per Induktion über ein magnetisches Feld ein kontinuierlicher elektrischer Strom erzeugen, dessen Spannung proportional zur Amplitude der Auslenkung der Membran und somit zur erzeugenden Schallwelle ist. Auf diesem Prinzip beruht die Funktionsweise des Mikrofons.

Analoge Signale lassen sich durch den *Lebesgueraum* $L^2(\mathbb{R})$ modellieren [57]. Dieser wird für eine Funktion $f: \mathbb{R} \to \mathbb{C}$ definiert durch

$$L^2(\mathbb{R}) := \left\{ f: \mathbb{R} \to \mathbb{C} \mid \text{f messbar und } \sqrt{\int_R |f(t)|^2 \, dt} < \infty \right\}.$$

Mittels eines Analog/Digital (A/D)-Wandlers lässt sich dieses analoge Signal in zwei Schritten *digitalisieren*, wie in Abbildung 2.2 illustriert ist. Dazu wird das Signal x zunächst in vorab festgelegten, diskreten Zeitabständen T abgetastet, indem diskrete Werte aus den Amplituden des Signalstroms extrahiert werden:

$$x_T(n) = x(Tn) \ , n \in \mathbb{Z}.$$

Die Anzahl $1/T$ der Abtastwerte (engl. *Samples*) je Sekunde wird als *Abtastfrequenz* oder *-rate* (engl. *sampling rate*) bezeichnet.

Bei einer solchen Wandlung stellt sich natürlich die Frage des Informationsverlustes. Dazu wird der Begriff der *Bandbegrenztheit* benötigt: Ein Signal $f \in L^2(\mathbb{R})$ heißt *bandbegrenzt*, falls eine

Frequenz $f_{\max}$ existiert, so dass das Spektrum $\hat{f}$ von f für alle Frequenzen außerhalb des Frequenzbandes $[-f_{\max}, f_{\max}]$ verschwindet, d.h. $\hat{f} = 0$ für alle Frequenzen $|f| > f_{\max}$.

Das Abtasttheorem von Shannon [94] besagt, dass ein bandbegrenztes Signal durch eine diskrete Menge von Abtastpunkten fehlerfrei reproduziert werden kann, wenn dieses mit einer Frequenz $f_A = 1/T$ abgetastet wird, die größer als doppelt so hoch ist, wie die höchste im abzutastenden Signal auftretende Frequenz $f_{\max}$, also

$$\frac{f_A}{2} > f_{\max}.$$

Dabei wird $f_A/2$ auch als *Nyquist-Frequenz* bezeichnet.

In dem der Abtastung nachfolgenden Prozess findet eine Quantisierung der Signalwerte auf Integerwerte eines fixen Wertebereiches statt, üblicherweise angegeben in Bits: Die Folge der Signalwerte wird entsprechend einem linearen oder nichtlinearen Schema auf Quantisierungswerte abgebildet. Die Anzahl der Abstufungen definiert somit die Qualität der Repräsentation: Gibt es zu wenige Abstufungen, so werden verschiedene Amplitudenwerte der selben Qualitätsstufe zugeordnet, was zu Quantisierungsrauschen führt. Telefonübertragung z.B. verwendet üblicherweise einen Wertebereich von 8 Bit, CD-Qualität dagegen 16 Bit, womit es 65536 Quantisierungsabstufungen gibt. Man unterscheidet hier zwischen linearer, also gleichverteilter, und nichtlinearer Quantisierung. Die am weitesten verbreitete Codierung digitaler Audiodaten ist *PCM* (engl. *pulse code modulation*).

Ein bandbegrenztes Audiosignal kann also anhand seiner Abtastwerte verlustfrei rekonstruiert werden, wenn die Abtastrate nur entsprechend hoch genug gewählt wird. Um eine solche Bandbegrenzung zu erreichen, wird das Signal vor der Wandlung üblicherweise durch einen Bandpass-Filter auf den für die geplante Abtastrate passenden Frequenzbereich begrenzt. Oft wird dieser durch den für das menschliche Ohr wahrnehmbaren Frequenzbereich von ca. 16Hz – 20 kHz [55] beeinflusst. Bei einer Audio-CD z.B. werden vor der Abtastung Frequenzen oberhalb von 22,05 kHz herausgefiltert und die Daten anschließend mit 44,1 kHz abgetastet.

Wird ein Signal mit einer Abtastrate diskretisiert, die geringer oder höchstens gleich dem Doppelten der höchsten im Signal enthalten Frequenz ist, so spricht man von *Undersampling*. In diesem Fall ist ein Informationsverlust unvermeidbar. Hinzu kommen neue, störende Klangbestandteile – *Aliasing* genannt. *Oversampling* hingegen führt auf der anderen Seite theoretisch nicht zu einem Mehr an Informationen, wird jedoch trotzdem gelegentlich verwendet, um z.B. Qualitätsverluste durch minderwertige Wandlerelemente auszugleichen.

Wir betrachten im Folgenden nur *digitale* Signale. Diese lassen sich allgemein durch den diskreten Lebesgue-Raums ℓ^2 geeignet modellieren [57]:
Für $I \subseteq \mathbb{Z}$ sei

$$\ell^2(I) := \left\{ x : I \mapsto \mathbb{C} \mid \sum_{i \in I} |x(i)|^2 < \infty \right\}.$$

Die Aufgabe der Audioidentifikation ist es, zu einem möglichst kurzen, in der Praxis wenige Sekunden langen Audiofragment (etwa $q \in \ell^2(\mathbb{Z})$ mit endlichem Träger) festzustellen, ob es Teil eines in einer Datenbank D vermerkten Audiodokumentes ist. Diese Datenbank enthält dabei Repräsentationen von Audiosignalen zusammen mit entsprechenden Metadaten wie Titel oder Copyright-Informationen. Da im nicht-trivialen Fall zu dem zu identifizierenden Audiosignal keinerlei beschreibende Metadaten mitgeliefert werden, besteht die Herausforderung darin, von den implizit enthaltenen Informationen auf die Identität des Audiofragmentes zu schließen. Daher spricht man auch von „*inhaltsbasierter Audioidentifikation*“ (engl. *content-based identification*, CBID). Falls die Identifikation erfolgreich ist, können aussagekräftige Metadaten zurückgeliefert werden.

Ein in der Praxis auftretendes Problem sind Filter bzw. Signalverzerrungen vielfältiger Art, denen Anfragesignale unterliegen:

- **Rauschen:** Dieses resultiert z.B. aus analoger Übertragung und Änderungen des Signal-zu-Rauschen-Verhältnisses (engl. *SNR*, *signal-to-noise ratio*).

- **Spektrale Veränderungen**: Oft werden beispielsweise im Radio- und Fernsehbereich *Equalizer*-Effekte eingesetzt, die bestimmte Frequenzbereiche anheben oder absenken, um auch trotz des verschiedenen Grundklangs abgespielter CDs einen Sender-eigenen Markenklang beizubehalten oder diesen an gängige Lautsprechersysteme anzupassen.

- **Änderungen des Dynamikverhaltens**: Eine gängige Methode, um z.B. als Radiosender in der Vielfalt moderner Beschallung aufzufallen, ist die Verwendung von *Kompressor*-Effekten. Dabei werden in Abhängigkeit von Schwellwerten Teile des Audiosignals kurzzeitig so verstärkt oder abgeschwächt, dass auf Grund des reduzierten Dynamikumfangs eine höhere Lautstärke erreicht werden kann.

- **Skalierung**: Eine weitere Methode, um Aufmerksamkeit zu erreichen, ist das Abspielen von Musikstücken mit leicht erhöhter Geschwindigkeit. Weiterhin kann der Effekt auch der wahrgenommenen Geschlossenheit eines Radioprogramms dienen. Dieses Prinzip ist darüber hinaus auch die Grundlage moderner Club-Tanzmusik, wo es intensiv eingesetzt wird, um den Übergang mehrerer Musikstücke ineinander trotz verschiedener Grundgeschwindigkeiten zu ermöglichen. Die Skalierung, auch *Pitching* genannt, führt zu Änderungen der Tonhöhe – im Extremfall zum sogenannten *Mickey-Mouse-Effekt*.

- **Translationseffekte:** Laufzeitechos oder -Verzögerungen (Halleffekte) verändern ein Audiosignal maßgeblich durch Einstreuen akustischer Reflektionen.

- **Andere Verzerrungen:** Diese können z.B. als Folge von De-/Kodierungseinflüssen wie der MP3-Kodierung oder durch Watermarking-Mechanismen entstehen. Weitere Ursachen sind physikalisch begründet, wie das Zusammenspiel von Lautsprecher, Übertragungseigenschaften des *akustischen Kanals* (der Luft) und Mikrofon (engl. „loudspeaker-microphone transmission“, *Ls-Mic*).

Auch unter solchen Einflüssen muss das Verfahren einen hohen Grad an Stabilität aufweisen, d.h. eine möglichst hohe Deformationstoleranz besitzen.
Da die genannten Signalveränderungen bei digitalen Signalen zu Bitänderungen auf Ebene einzelner Samples führen, macht ein samplegenauer *Brute-Force*-Vergleichsmechanismus von Anfrage- und Datenbankdokument keinen Sinn. Hinzu kommt die Größe von Audiodokumenten, die in erster Linie durch die auf menschliches Hörempfinden hin gewählten hohen Abtastraten entstehen: Bei CD-Qualität beispielsweise beträgt die aus der Multiplikation von Abtastrate, Werte-Auflösung in Bit und Kanalanzahl entstehende Größe ca. 172 kb/s, d.h. ein einziges Musikdokument mit einer Länge von drei Minuten hat eine Audiodatengröße von rund 30MB. Vergleiche in solchen Dimensionen sprengen aktuelle Rechenleistungsgrenzen bei Weitem, vor allem im Hinblick auf die Kapazität von Audio-Datenbanken, die aktuell bis zu 10^6 Dokumente enthalten.

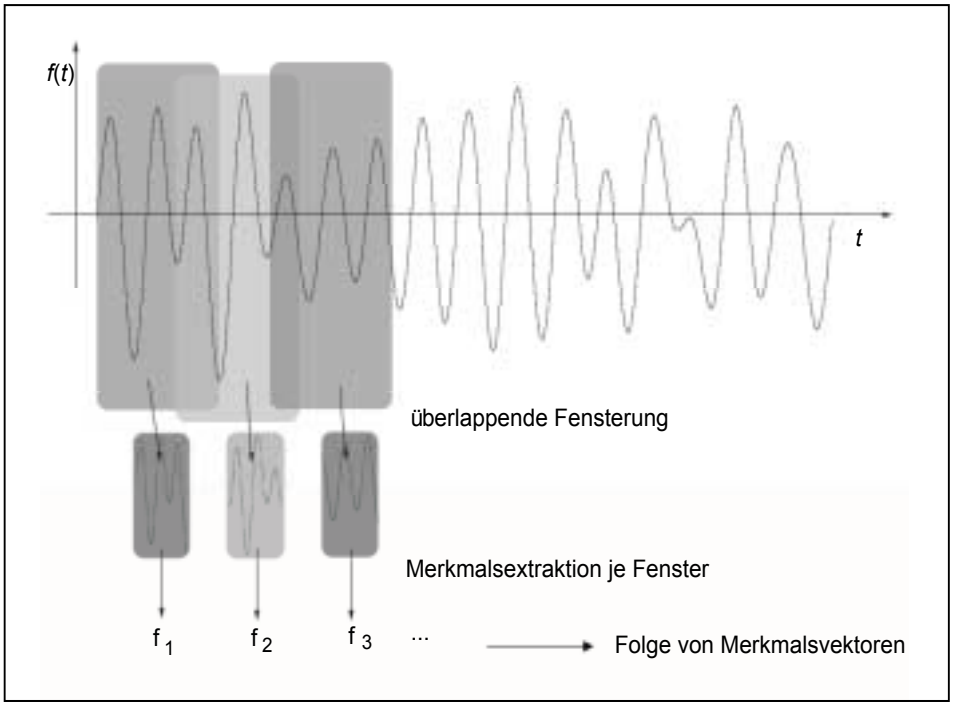


Abbildung 2.3: Prinzip der Merkmalsextraktion

2.3 Der Audiofingerabdruck

Um trotz dieser Herausforderungen eine nahezu echtzeitfähige Erkennung zu erreichen, arbeiten die gängigen Audioidentifikationsverfahren nicht auf den eigentlichen Audiodaten, sondern auf daraus extrahierten Audiomerkmalen (engl. *audio features*). Für ein Signal

$$x \in \ell^2(\mathbb{Z}) \subset \mathbb{Z} \times \mathbb{R}$$

liefert ein Merkmalsextraktor F eine per Transformation erzeugt Merkmalsmenge

$$F[x] \subset \mathbb{Z} \times X,$$

wobei X eine im Allgemeinen endliche Menge von Merkmalsklassen ist.
Ein einfaches Beispiel für $X := \{0,1\}$ stellt folgender Merkmalsextraktors dar:

$$F[x](t) = \begin{cases} 1, \text{ falls } x \text{ ein lokales Maximum an Stelle t besitzt} \\ 0 \text{ sonst} \end{cases}$$

Üblicherweise sind vor allem endliche Signale $x : [a,b] \to \mathbb{R}$, $a \le b$, a und b ganzzahlig, von Interesse. Die Merkmalssignale folgen dann einer Abbildung $I \to X$, wobei $I \subset \mathbb{Z}$ eine endliche Teilmenge ist, welche die Positionen der Merkmale angibt [57]. Dazu wird meist eine sogenannte *Fensterfunktion* verwendet, die dem Merkmalsextraktor einen transformierten Fenster-Ausschnitt des Signals liefert. Der Extraktor berechnet zu diesem Fenster ein repräsentatives Merkmal, oft dargestellt durch einen Vektor f_i. Anschließend wird das Fenster schrittweise um eine fixe Einheit verschoben. Auf Fensterfunktionen wird in Abschnitt 2.4.1b genauer eingegangen. Aufgrund der Regelmäßigkeit der verwendeten Fenster spricht man in diesem Fall von *äquidistanten* Merkmalen. Im *nicht-äquidistanten* Fall werden Merkmale dagegen *ereignisbasiert*, also ohne festes Zeitraster erzeugt. Der Prozess der Merkmalsextraktion ist in Abbildung 2.3 skizziert.

Der so erzeugt *Audiofingerabdruck* [21] (engl. *audio fingerprint)* ist eine Folge solcher Merkmalsvektoren und stellt somit eine kompakte Repräsentation relevanter Teile des ursprünglichen Audiosignals dar [32]. Der Audiofingerabdruck kann für einen schnellen und ressourcenschonenden Vergleich mit in einer Datenbank gespeicherten Audiofingerabdrücken unter Einbeziehung eines Distanzmaßes verwendet werden. Der Merkmalsextraktor F sollte dabei robust gegenüber Transformationen sein, die das Signal in einem gewissen Rahmen verändern, ohne es vollkommen zu verfremden. Idealerweise sollte F in der Lage sein, zu dem selben Audiosignal auch nach verschiedenen moderaten Signaltransformationen einen möglichst identischen Fingerabdruck zu generieren.

Das Audiofingerabdruck-Prinzips besitzt einige Vorteile:

- **Formatunabhängigkeit:** Der Audiofingerabdruck ist rein inhaltsbasiert und daher unabhängig von Dateiformaten.

- **Keine Daten außer dem Audiosignal zur Erzeugung nötig**: Der Audiofingerabdruck wird durch Verwendung impliziter Semantik im Audiosignal generiert.

- **Kompakte Speicherung**: Im Vergleich zum Audiosignal ist der Platzbedarf geringer.

- **Zentrale Metadaten**: Diese können dadurch effektiver gepflegt und nach Identifikation eines Stückes mit diesem verlinkt werden.

- **Effektiver Vergleich**: Da nur relevante Informationen vorhanden sind, kann ein Vergleich mit anderen Audiofingerabdrücken mit geringem Aufwand durchgeführt werden.

Demgegenüber sind folgende Nachteile des Fingerabdruck-Prinzips zu nennen:

- **Begrenzte semantische Ausdruckskraft**: Die automatisch extrahierten Merkmalswerte besitzen eine lediglich geringe Semantik. Man spricht auch von der „semantischen Lücke" (engl. *semantic gap*) zur menschlichen Inhaltsbeschreibung.

- **Mögliche Ähnlichkeit**: Die Abstraktion kann dazu führen, dass aus verschiedenen Audiosignalen identische Merkmale erzeugt werden. Dies kann zu falschen Identifikationen (engl. *False Positives*) führen.

Ein der Merkmalsextraktion verwandtes Konzept ist die *Merkmalsselektion*. Diese arbeitet auf dem Raum der Merkmale und reduziert dessen Dimensionalität durch Ausschluss von Merkmalsinformationen, etwa durch Analyse der Merkmale bezüglich der *intrinsischen Dimensionalität* der Merkmale. Dieser Ausdruck beschreibt die Eigenschaft, dass bestimmte d-dimensionale Datenmengen adäquat in einem m-dimensionalen Subraum mit Dimensionalität $m<d$ dargestellt werden können [52], z.B. Datenvektoren im $\mathbb{R}^3$, die auf einer Linie liegen und somit im R^1 dargestellt werden können. Da jedoch eine exakte Unterteilung eines Prozesses in Extraktion und Selektion oftmals nicht möglich ist, werden nachfolgend beide Begriffe als Extraktion zusammengefasst.

2.4 Ein generisches Audioidentifikationssystem

Ein auf Audiofingerabdrücken arbeitendes Identifikationssystem besitzt im Allgemeinen zwei wesentliche Komponenten: Den Merkmalsextraktor und den Vergleichsmechanismus mit einer Audiodatenbank.
Für den Entwurf eines Merkmalsextraktor ergeben sich bestimmte Herausforderungen:

- **Einfache Berechenbarkeit der Merkmalsfolgen:** Um eine effektive Verarbeitung zu ermöglichen, sollte das Erzeugen der Merkmalsfolgen in Echtzeit möglich sein.

- **Reduktion der möglicherweise hohen Dimensionalität des Vektorraums der erzeugten Merkmalsfolge:** Je nach Wahl des Merkmalsextraktors kann die Dimensionalität durch nachfolgende Transformationen weiter reduziert werden, um eine stärkere Kompaktheit des Fingerabdrucks zu erreichen. Hier findet ein Trade-Off zwischen einer Reduktion der Dimensionalität und dem damit verbundenen Informationsverlust statt.

- **Nötige Varianz eines Fingerabdrucks zu anderen Fingerabdrücken:** Perzeptuell ähnliche Audiosignale müssen gut genug unterscheidbar sein, verschiedene Repräsentationen des selben Signals müssen jedoch übereinstimmen. Dies muss insbesondere auch dann möglich sein, wenn mit großen Mengen von Audiofingerabdrücken gearbeitet wird.

- **Robustheit:** Der Merkmalsextraktor muss gegenüber den oben genannten Signalstörungen in möglichst hohem Maße robust sein. Ebenso sollten Signalparameter wie Lautheit oder Abspielgeschwindigkeit keinen Einfluss auf die erzeugte Merkmalsfolge besitzen.

- **Dichtheit der erzeugten Merkmalsfolge:** Regelmäßige, innerhalb eines zu definierenden Toleranzbereiches nah beieinander liegende Merkmalsvektoren sind über die gesamte Signaldauer zu garantieren, um eine möglichst gleich gute Verwendbarkeit des Fingerabdrucks zu jedem Zeitpunkt zu gewährleisten.

Auch der Vergleichsmechanismus unterliegt verschiedenen Anforderungen:

- **Effektive Berechnung der Distanz zweier Merkmale:** Das für den Vergleich von Audiofingerabdrücken im Raum der Merkmale eingesetzte Distanzmaß muss ebenso schnell und korrekt arbeiten – d.h. keine Treffer auslassen und eine geringe *False Rejection Rate* (FRR) besitzen – wie speichereffizient und leicht pflegbar sein [11].

- **Effizientes Suchen**: Dies kann z.B. durch Verwenden von Indizes erreicht werden, die eine Reduktion der Dimension der Merkmale ermöglichen. Dadurch kann dem „Curse Of Dimensionality“ [16] – dem mit steigender Dimensionalität exponentiell steigenden Rechenaufwand – entgegengewirkt werden. Eine entscheidende Rolle in Anbetracht stetig steigender Datenbestände spielt die Skalierbarkeit des zugrunde liegenden Datenbanksystems.

- **Kompakte Identifikation:** Die Erkennung eines Audiostückes sollte anhand eines nur wenige Sekunden langen Signalstückes möglich sein.

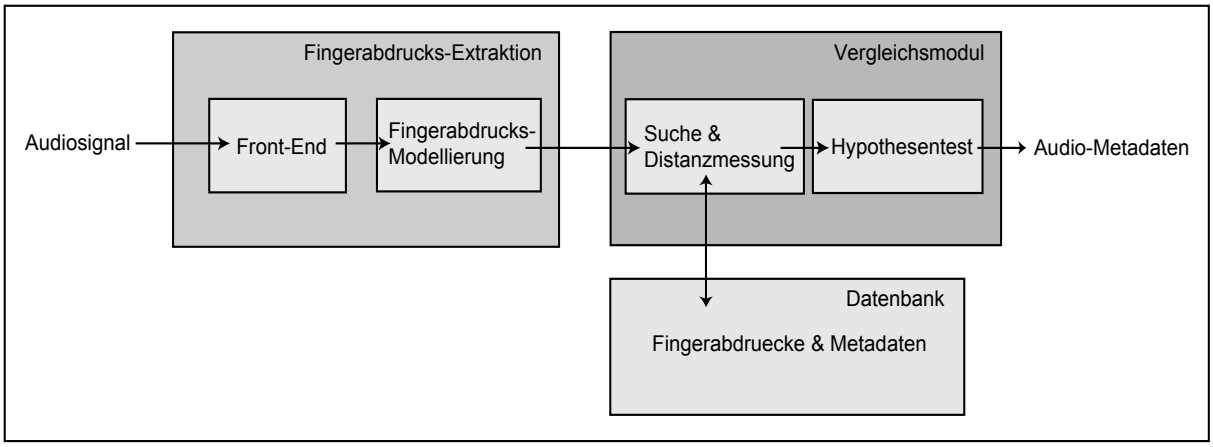


Abbildung 2.4: Struktureller Aufbau eines Audioidentifikationssystem

Audioidentifikationssysteme unterscheiden sich im Wesentlichen durch die Art, Verwendung und Speicherung von Merkmalen, die Wahl eines Ähnlichkeitsmaßes und die Handhabung des Vergleiches von Anfrage und Datenbank [13]. Die wissenschaftlichen Hintergründe aktueller Systeme umfassen neben an Mustererkennung und dem klassischen Multimedia-Retrieval angelehnte Verfahren auch Ansätze aus Sprachverarbeitung [22], Kryptographie [45] und Bioinformatik [22]. Nichtsdestotrotz weisen alle bislang vorgestellten Methoden strukturelle Gemeinsamkeiten auf.
In Anlehnung an Cano et al. [21] wird ein Audioidentifikationssystem in zwei Module aufgegliedert, wie in Abbildung 2.4 dargestellt ist:

1) ein Modul zur Fingerabdrucksextraktion, bestehend aus *Front-End* und *Fingerabdrucksmodellierer*

2) ein Vergleichsmodul mit Submodulen *Datenbanksuche*, *Distanzmessung* und *Hypothesentest*

Die einzelnen Module werden nachfolgend erläutert, wobei nach Möglichkeit auf die in gängigen Audioidentifikationssystemen verwendete Realisierung von Teilmodulen eingegangen wird.

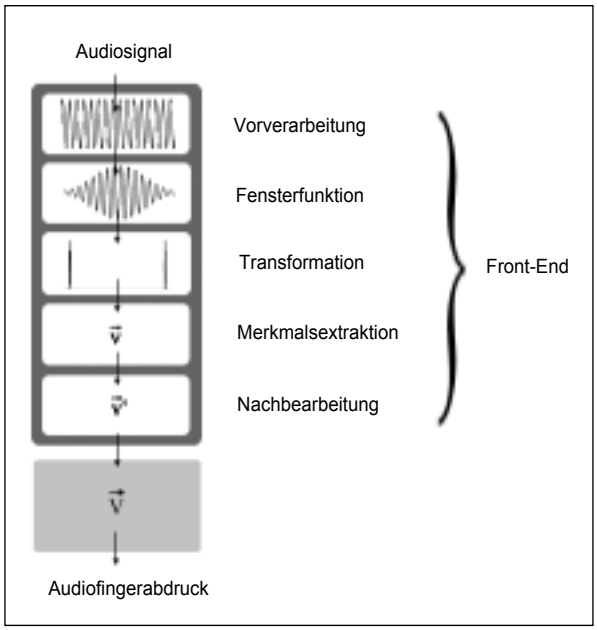


Abbildung 2.5: Audiofingerabdrucks-Extraktion

2.4.1 Front-End

In diesem seriellen fünfstufigen Teilsystem, illustriert in Abbildung 2.5, wird das zu identifizierende Audiosignal in eine Sequenz von Merkmalen konvertiert, welche dem Modelliermodul als Eingabe dienen. Hauptaufgaben sind neben der Reduzierung der Dimensionalität das Extrahieren perzeptuell relevanter Daten und die Gewährleistung der Robustheit.

a) Vorverarbeitung

Das eingehende Audiosignal wird für die weitere Bearbeitung in ein vorab gewähltes, einheitliches Format konvertiert. Die dazu durchgeführten Transformationen umfassen unter anderem:

- Analog-/Digitalwandlung, kurz *A/D-Wandlung*
- Umrechnung mehrkanaliger Datenströme, z.B. stereo nach mono
- Anpassung/Umrechnung der Abtastrate (engl. *resampling*): Es wird unterschieden zwischen Up- und Downsampling, je nachdem, ob die Abtastrate erhöht oder verringert wird. Beides wird durchgeführt mittels verschiedener Interpolationsalgorithmen wie z.B. linearer und bandbegrenzter Interpolation und Filterung [96].
- Vorverstärkung
- Normalisierung, d.h. anteiliges Skalieren des Wertebereichs des Signals auf den möglichen digitalen Wertebereich
- Bandpassfilterung: Ausschließen von Frequenzbereichen, z.B. als Vorverarbeitung für eine A/D-Wandlung
- Dekodieren/Kodieren, z.B. Umwandeln in das MP3-Format.

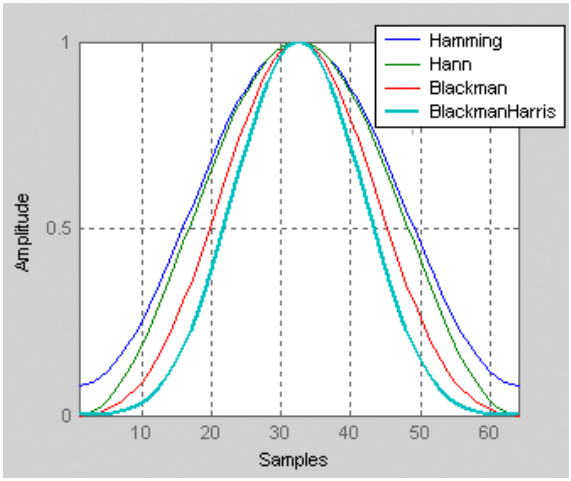
b) Fensterfunktion

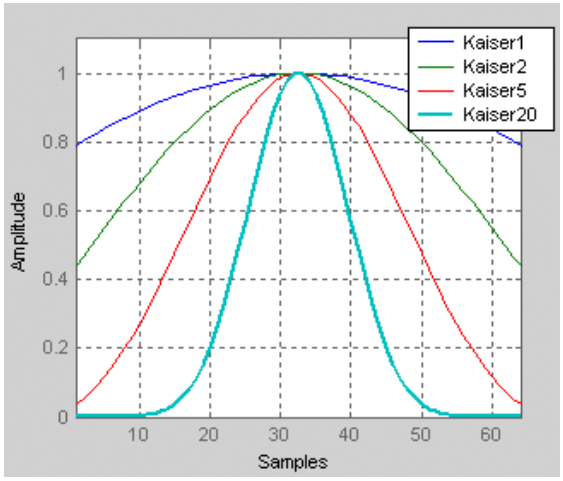
Die Verwendung einer Fensterfunktion erlaubt eine blockweise Bearbeitung des Eingangssignals, wobei sich die entstehenden *Frames* üblicherweise zu einem gewissen Grad überlappen, um Informationsverlust an den Blockenden zu verhindern. Die Fensterfunktion selbst strebt an den Blockenden gegen 0 und ist achsensymmetrisch. Auf diese Weise werden auch Diskontinuitäten vermieden. Grund für den Einsatz einer Fensterfunktion ist die für nachfolgende Raum-Frequenzraum-Transformationen zeitliche Eingrenzung des Signals.

Um zu einen Zeitpunkt t aus einem Signal $x(r)$ einen Frame $x'(r)$ zu erhalten, werden das Signal und die Fensterfunktion punktweise miteinander multipliziert:

$$x'_t(r) := x(r) \cdot w(r - t).$$

Es sei angemerkt, dass t je nach Anwendung den Start- (linke Kante des Fensters) oder Mittelpunkt der Fensterposition bezeichnet. Die Länge des Fensters spielt darüber hinaus eine wichtige Rolle: Ein zu kleines Fenster erlaubt keine relevanten Aussagen über tieffre-





2.6a: Die generalisierte Kosinusfuntkion (links), Abb. 2.6b: Die Kaiserfunktion mit verschiedenen *a*-Werten (rechts)

quente Signaländerungen, während sich mit steigender Länge des Fensters auch die Länge des damit korrespondierenden Signalausschnitts und damit die Wahrscheinlichkeit erhöht, dass sich im Signalausschnitt wesentliche spektrale Veränderungen befinden. Dies führt dazu , dass die Spektralanalyse keine gute Auflösung für hohe Frequenzen besitzt.

Im Folgenden sind gängige Fensterfunktionstypen aufgeführt [81]. Jede Funktion liefert einen Frame *W* der Länge *n*. Durch verschiedene Parameterwerte entstehen Variationen der jeweiligen Fensterfunktion, die teilweise sogar eigene Bezeichnungen besitzen. Im Folgenden gilt für die Fensterfunktion *w* stets $w(k) = 0$ für $k<0$ oder $k>n$.

i. Rechteck / Boxcar

Diese Funktion liefert ein rechteckiges Fenster zurück und ist nur der Vollständigkeit halber aufgeführt. Das Verwenden der Boxcar-Funktion entspricht dem Arbeiten ohne Fensterfunktion.
Es gilt $w(k) = 1$, falls $k \in \{0,...,n-1\}$.

ii. Generalisierte Kosinus-Fensterfunktion / Hamming / Hann / Blackman

Hierbei handelt es sich um eine populäre Familie von Fensterfunktionen, die einer gemeinsamen Form folgen und in Abbildung 2.6a dargestellt und wie folgt für $k \in \{0,..n\}$ definiert sind:

$$w(k+1) = a_0 - a_1 \cdot \cos\left(2\pi \frac{k}{n-1}\right) + a_2 \cos\left(4\pi \frac{k}{n-1}\right) - a_3 \cos\left(6\pi \frac{k}{n-1}\right)$$

Die jeweiligen Parameter a_i sind dabei der Tabelle 2.6c zu entnehmen.

Funktionstyp	**a_0**	**a_1**	**a_2**	**a_3**
Hamming	0,54	1- a_0	0	0
Hann	0,5	1- a_0	0	0
Blackman	0,42	-0.5	0,08	0
Blackman-Harris	0,35875	0,48829	0,14128	0,01168

Tabelle 2.6c

iii. Kaiser

Die Form dieser Funktion kann durch verschiedene Werte des reellwertigen Parameters *a* verändert werden: Je größer die Werte von $|a|$, desto enger wird das Fenster. Ein Wert von 0 korrespondiert mit der Rechtecksfunktion, wohingegen sich das Fenster für steigende $|a|$ immer mehr an eine Gausskurve annähert. Dazwischen werden die Formen verschiedener Fensterfunktionen approximiert, z.B. Blackman durch den Wert *a* = 8,885. Diese Funktion ist in Abbildung 2.6b dargestellt.

$$w(k) = \frac{I_0(\pi \cdot a\sqrt{1-(\frac{2k}{n-1})^2})}{I_0(\pi \cdot a)}$$

Dabei ist I_0 die modifizierte Besselfunktion 0-ter Ordnung.

Neben weiteren Fenstertypen wie Chebyshev, Bartlett und der verwandten Dreiecksfunktion gibt es natürlich auf spezielle Zwecke zugeschnittene Fensterfunktionen, wie z.B. die nachfolgend aufgeführten, die in verschiedenen Audio-Codecs Anwendung finden [116]:

iv. MP3 / MPEG-2 AAC

$$w(k) = \sin\left[\frac{\pi}{2n}(k+\frac{1}{2})\right]$$

v. Ogg-Vorbis

$$w(k) = \sin\left[\frac{\pi}{2}\sin^2\left(\frac{\pi}{2n}(k+\frac{1}{2})\right)\right]$$

c) Transformation

In dieser Verarbeitungsstufe wird die Dimensionalität des Signals verringert. Ziel ist die Reduzierung der Redundanz durch vergrößerte Abstrahierung in Form von Kompaktheit, d.h. Reduzierung der Dimension ohne Verlust an intrinsischer Information. Üblicherweise werden hierzu lineare Transformationen verwendet.

Gegeben sei eine $n \times d$-dimensionale Matrix *X* aus *n* *d*-dimensionalen Datenvektoren. Die lineare Transformation von *X* in eine $n \times m$-Zielmatrix *Y*, wobei hier $m<d$, ist allgemein definiert als $Y = HX$, wobei H die $d \times m$-Matrix der linearen Transformation ist.

Zwar existieren optimale lineare Transformationen, wie z.B. die Hauptkomponentenanalyse (engl. *principal component analysis, PCA*), die jedoch oft sehr rechenintensiv sind. Obwohl Fortschritte in Algorithmik und Hardware immer komplexere Berechnungen erlauben, werden meist jedoch weniger komplexe Transformationen, vor allem *Spektraltransformationen* (Transformationen aus dem Zeit- in den Frequenzraum), verwendet. Mit Hilfe dieser Transformationen wird zum einen Rauschen entfernt und zum anderen eine komprimierte Datenrepräsentation erreicht. Weiterhin beruhen in den meisten Fällen die nachfolgenden Verarbeitungsstufen auf bestimmten Eigenschaften der Transformation. Auch in diesem Fall geht ein Gewinn an Dimensionsreduktion je nach Güte der Transformation einher mit einem Verlust an Information.

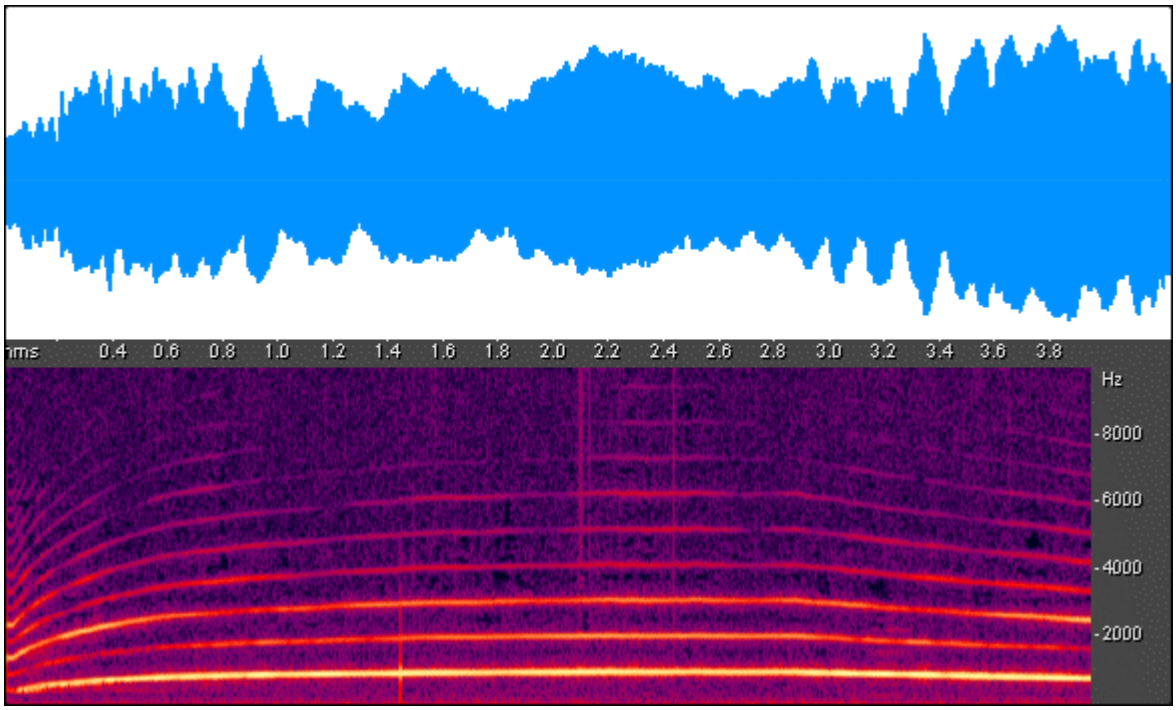


Abbildung 2.7: Wellenform und Spektrogramm einer Sirene, visualisiert mit Cooledit Pro.

Die gebräuchlichste Spektraltransformation ist die Fourierentwicklung: Periodische Signale *x(r)* mit einer Periodendauer T_0 lassen sich in eine Superposition von Sinus- und Kosinusfunktionen mit Grundfrequenz $f_0 = 1/T_0$ entwickeln:

$$x(r) = a_0 + \sum_{n=1}^{\infty} a_n \cdot \cos(n \cdot w_0 \cdot r) + b_n \cdot \sin(n \cdot w_0 \cdot r).$$

Dabei ist $w_0 = \frac{2\pi}{T_0}$. In dieser Fourierreihe sind a_n, b_n als *n*-te Fourierkoeffizienten das Maß für die Anteile, mit denen die einzelnen Sinus- und Kosinusfunktionen zum Gesamtsignal *x(r)* beitragen.

Zur Berechnung der Fourierentwicklung wird für (diskrete) endliche Signale der Länge *m* üblicherweise die sogenannte *schnelle Foutiertransformation* (engl. *Fast Fourier Transform*, *FFT*) verwendet, die in Rechenzeit $O(m \log m)$ berechnet werden kann.

Da das fouriertransformierte Signal keine zeitliche Informationen enthält, wurde die *gefensterte* (diskrete) schnelle Fouriertransformation (engl. *Short Time Fourier Transformation*, *STFT*, manchmal auch *Windowed Fourier Transformation*, *WFT*) entwickelt, die für Fensterfunktionen mit Träger berechnet werden kann. Die *STFT* transformiert ein gefenstertes Signal *x(r)* der Fensterlänge N in Rechenzeit $O(N \log N)$ und ermöglicht auf diese Weise die zeitliche Zuordnung spektraler Informationen.

Das Betragsquadrat der STFT wird als diskretes *Spektrogramm* von *x* bezeichnet:

$$Spec_x(t,f) := |STFT_x(t,f)|^2 = \left| \sum_{n=0}^{N-1} x(n-t) \cdot w(n) \cdot e^{-2\pi i \cdot \frac{ft}{N}} \right|^2$$

Das Spektrogramm beschreibt also die Faltung von *x* und der Impulsantwort *w* eines Filters zur angegebenen Frequenz *f*. Somit stellt es eine Zusammenführung der Frequenzmessungen einer Reihe einzelner Filter dar, deren Frequenzbänder um ein Vielfaches *k* der Grundfrequenz f_o zentriert sind: $f_k = k \cdot f_0$, wobei *k* die Centerfrequenz des *k*-ten Frequenzbandes ist. Eine beispielhaftes Spektrogramms ist in Abbildung 2.7 dargestellt.

Die positiv definierte Funktion $Spect_x(t, f)$ kann auch probalistisch als unnormalisierte Wahrscheinlichkeitsdichtefunktion (WDF) über die Frequenz *f* interpretiert werden, nämlich als Wahrscheinlichkeit, dass eine Frequenz in einem Frame vorkommt. Zwar ist die Frequenz keine Zufallsvariable, jedoch erlaubt die Einführung dieser Interpretation in nachfolgenden Verarbeitungsstufen die Anwendung von Konzepten aus der Wahrscheinlichkeitstheorie.

Die Hauptkomponentenanalyse (PCA [83], auch als *Karhunen-Loeve*-Transformation (*KL*) bekannt) ist wie oben erwähnt relativ aufwendig zu berechnen. Es existieren allerdings praxistaugliche Approximationen und Varianten. PCA ist eine lineare Projektionstechnik und ermöglicht durch eine lineare Transformation $M: \mathbb{R}^d \rightarrow \mathbb{R}^k$ die Repräsentation von *d*-dimensionalen Daten durch eine niedriger dimensionierte Datenmenge mit Dimensionalität *k*, $k < d$. Erreicht wird dies durch sukzessive Bestimmung von „Koordinatenachsen" in der Richtung der größten Varianz der Datenmenge. Richtungen verschwindender Varianz werden vernachlässigt, wodurch eine kompaktere Darstellung der Datenvektoren in reduzierter Dimension erreicht wird.

Die PCA ist optimal im Sinne des mittleren quadratischen Fehlers: Gegeben eine Menge von Vektoren $x_i \in \mathbb{R}^d,\ i \in \{1..N\}$ minimiert die PCA *M* den entstehenden quadratischen Fehler

$$e = \sum_{i=1}^{N} (x_i - M^T M x_i)^T (x_i - M^T M x_i).$$

Die PCA weist noch weitere interessante Eigenschaften auf, auf die hier aber nicht weiter eingegangen werden soll. Interessierte Leser seien an Bishop et al. [17] verwiesen. Durch die PCA lässt sich somit die Dimensionalität ohne großen Verlust an signifikanten Daten reduzieren. Das Maß an Reduktion hängt davon ab, wie die in den Originaldaten enthaltenen Informationen verteilt sind. Eine effizientere Variante, die „orientierte Hauptkomponentenanalyse" (OPCA) wurde in [20] erfolgreich für die Extraktion rauschrobuster Audio-Merkmale angewandt.

Die diskrete Kosinus-Transformation (engl. *discrete cosine transformation, DCT*) stellt eine der diskreten Fouriertransformation ähnliche Transformationsfamilie aus mehreren Varianten dar, arbeitet allerdings im Gegensatz zu dieser mit reellen Daten. Verwandt sind weiterhin die Diskrete Sinus-Transformation und die modulierte DCT, welche auf überlappenden Datenblöcken arbeitet. Die DCT besitzt eine *Frequenzkompaktheit* genannte Eigenschaft, die sich darin äußert, dass ein Großteil der Signalinformation in wenigen Komponenten der DCT konzentriert wird [116]. Unter bestimmten Bedingungen approximiert die DCT daher die PCA. Inspiriert von der erfolgreichen Anwendung der DCT in der Arbeit mit Sprachdaten fanden Logan et al. [63] heraus, dass sie auch für die Arbeit auf Audiodaten gut geeignet ist. Die DCT kann wie die FFT in Zeit $O(n \log n)$ berechnet werden.

Darüber hinaus existieren noch weitere lineare Transformation wie die Haar-Transformation oder Wavelets, auf die hier aber nicht weiter eingegangen werden soll.

In manchen Systemen werden nach der folgenden Merkmals-Extraktion noch weitere lineare Transformationen durchgeführt, um die Dimensionalität weiter zu reduzieren oder um eine Glättung der Vektordaten zu erreichen (z.B. DCT bei der Verwendung von MFCCs, die im nachfolgenden Abschnitt erläutert werden [63]).

d) Merkmalsextraktion

In diesem Teilschritt des Front-Ends wird aus dem vorverarbeiteten und transformierten Eingangssignal mittels eines Merkmalsextraktors eine Folge von Merkmalen erzeugt. Die dazu verwendeten Ansätze unterscheidet man in parametrische und nicht-parametrische Methoden: Im parametrischen Fall folgen alle Signale einem mathematischen statistischen Modell und unterscheiden sich lediglich in den Modellparametern. Ein solches Modell könnte beispielsweise aus mehrdimensionalen Gauß-Verteilungen von Merkmalen bestehen und die Parameter aus den jeweiligen Mittelpunkten und Varianzen. Da ein solches Modell in der Praxis jedoch wegen der großen Signalvielfalt kaum findbar und falls doch, die Modellparameter nur sehr aufwendig bestimmbar sind, verwendet man üblicherweise nicht-parametrische Methoden [27].

Die Menge der Audiomerkmale lässt sich weiterhin in *semantische* und *nicht-semantische* Merkmale unterteilen. Andere Quellen sprechen auch von *wahrnehmbaren* respektive *physikalischen* Merkmalen. Semantische Merkmale bezeichnen auch für Menschen intuitiv erkennbare Bewertungen wie z.B. *Genre* oder *Beats-Per-Minute* (BPM). Semantische Merkmale weisen jedoch wesentliche Nachteile auf:

- Sie sind oft mehrdeutig, subjektiv und/oder inkonsistent über mehrjährige Betrachtung (z.B. *Genre*).
- Sie sind meist komplexer zu berechnen als nicht-semantische Merkmale.
- Sie sind nicht universell anwendbar. Bei bestimmten Musikrichtungen lässt sich beispielsweise die BPM-Zahl nicht bestimmen.

Nachfolgend wird daher nur auf nicht-semantische Merkmale eingegangen.

Obwohl sich die verschiedenen Fingerabdrucksverfahren in ihrem Hintergrund und der verbundenen Zielsetzung unterscheiden, beruhen viele auf den selben Merkmalsextraktoren, oft basierend auf dem Spektrogramm des Eingangssignals. Dabei ist es durchaus üblich, eine Reihe von Merkmalen zu kombinieren, um die Robustheit gegenüber störenden Einflüssen zu erhöhen. Häufig wird Vorwissen aus der *Psychoakustik* zu Eigenarten des menschliches Gehörs verwendet, um eine Datenreduktion zu erreichen.

Oft wird das in den Frequenzraum transformierte Signal in Bänder aufgeteilt, um z.B. den Eigenheiten des nichtlinearen menschlichen Gehörs Rechnung zu tragen und bestimmte Frequenzanteile stärker im Extraktionsprozess zu gewichten. Neben linearen und logarithmischen Bandanordnungen seien hier vor allem die Bark- und die Mel-Skalierung genannt. Die Bark-Skala besitzt üblicherweise 24 Bänder, entsprechend den ersten 24 kritischen Bändern, in welche die für menschliches Hören zuständige Basilarmembran im Innenohr eingeteilt werden kann [95]. Die Mel-Skala ist linear bis 1 kHz und nachfolgend logarithmisch. Daher stellt ausgehend von 1 kHz, denen ein Wert von 1000 Mel zugeordnet ist, jeder Wert m der Mel-Skala diejenigen Frequenz f dar, die $q(m) = m \cdot \frac{Hz}{Mel}$ mal so hoch wahrgenommen wird [63]. Durch beide Skalen werden Frequenzen in höheren Frequenzbereichen stärker zusammengefasst.

Die folgende – keineswegs erschöpfende – Auflistung liefert einen Überblick über wichtige klassische Audiomerkmale [21, 27, 115]. Diese entstammen ursprünglich verschiedenen technischen Anwendungsszenarien, vor allem statistischer und akustischer

Art, wie der Signalverarbeitung von Sprachsignalen bzw. der Diskriminierung von Sprache und Musik. Die Berechnung der einzelnen Merkmale erfolgt im Allgemeinen framebasiert.

Spektrale Merkmale:

- Lautheit (engl. *volume*, *loudness, short time energy*):
  Beschreibung der Signalamplitude („Energie“) zu einem Zeitpunkt t unter Verwendung einer Fensterfunktion $w(r)$, äquivalent berechenbar sowohl im Spektral- als auch im Zeitbereich:

$$Vol(t) := \frac{\int Spect_x(t,f)df}{\int w(r)dr} = \frac{\int |\, x(r)w(r-t)\,|^2\, dr}{\int w(r)dr}$$

  für $\int w(r)dr \neq 0$.

- Bandenergie:
  Energiebetrag in einem Frequenzband $[f_0, f_1]$ zu einem Zeitpunkt t unter Verwendung einer Fensterfunktion $w(r)$:

$$BE_{[f_0,f_1]}(t) := \frac{\int_{f_0}^{f_1} Spect_x(t,f)df}{\int w(r)dr}.$$

  Haitsma et al. [45] verwendeten in 33 Frequenzbändern die Vorzeichen der Differenz jeweils zweier benachbarter Bänder, um einen 32 Bit langen Hashwert zu generieren. Verschiedene Signaltypen können aufgrund ihrer spektralen Eigenheiten gezielt charakterisiert werden. Sprachsignale z.B. weisen in mehreren Bereichen des Spektrums deutliche charakteristische Eigenschaften auf.

- Bandenergieanteil (engl. *band energy ratio*):
  Anteil der Signalenergie eines Frequenzbandes $[f_0, f_1]$ an der Gesamtenergie in einem Frame t:

$$BER_{[f_0,f_1]}(t) := \frac{BE_{[f_0,f_1]}(t)}{Vol(t)}.$$

- Mittelfrequenz (engl. *median frequency* oder *centroid frequency*):
  Amplitudengewichteter Mittelwert des Spektrums in einem Frame t:

$$MedF(t) := \frac{\int f \cdot Spect_x(t,f)df}{\int Spect_x(t,f)df}.$$

  Dies kann auch als erstes statistisches Moment der WDF in einem Frame t interpretiert werden – was genau den Mittelwert definiert.

- Bandbreite:
  Die Varianz (zweites statistisches Moment) des als WDF interpretierten Spektrums eines Frames *t* ist ein Maß für die im Signal enthaltene Frequenzstreuung. Je größer diese ist, desto höher ist die Bandbreite:

$$BndW(t) := \frac{\int [f - MedF(t)]^2 \cdot Spect_x(t,f)df}{\int Spect_x(t,f)df}.$$

- Cepstrum / Cepstral-Koffizienten:
  Fouriertransformierte des gefensterten Frequenzspektrums eines Frames *t*, also gewissermaßen das Spektrum einer Zeitspalte des *STFT*-Spektrums. Die Bezeichnung *Cepstrum* selbst ist eine Permutation des Wortes *Spectrum*. Zur folgenden Berechnung der *Cepstral-Koeffizienten Cep*(*t*,*r*) wird angenommen, dass das Spektrum eines Frames wieder periodisch ist [115]:

$$Cep(t,r) := \frac{1}{2\pi} \int_{-\pi}^{+\pi} \log |Spect_x(t,f)| \cdot e^{i \cdot 2\pi \cdot f \cdot r} df.$$

  Die Variable *r* steht dabei für die Frequenz, in der sich Frequenzen im Spektrum periodisch wiederholen. Einheit ist das Wortspiel *quefrency*, die Frequenz einer Frequenz, also eine Zeiteinheit. Die durch Anwendung kleiner *r* entstehenden *ersten* Cepstral-Koeffizienten beschreiben die Ausprägung niederfrequenter Schwingungsanteile im Spektrum, die Koeffizienten höherer *r* die Ausprägung hochfrequenter Anteile. Auf diese Weise kann die „Glattheit" des Spektrums gemessen werden.

  Das Verfahren stammt aus der Sprachverarbeitung, wo es zur Fundamentalfrequenzschätzung eingesetzt wird, um über die ersten Cepstrum-Koeffizienten Informationen bestimmter Sprachbestandteile (Phonemformanten) zu gewinnen bzw. zu verändern. Die Verwendung des Logarithmus in der Formel hat dort vereinfachende Gründe, darüber hinaus jedoch als positiven Nebeneffekt eine Anpassung an menschliche Hörfähigkeiten: Bei gleicher Amplitude werden niedrigere Frequenzen näherungsweise logarithmisch leiser wahrgenommen als höherfrequente Anteile.

  Eine Erweiterung des Verfahrens stellt die Mel-Skalierung des Spektrums dar, die vor der Erstellung des Cepstrums durchgeführt wird. Entsprechend heißen die in diesem Zusammenhang verwendeten Koeffizienten *MFCC*s („Mel-frequency cepstral coefficients"). Diese werden z.B. in [22] unter Einbeziehung der DCT verwendet.

- Pitch / Grundfrequenz:
  Dieses Merkmal bezeichnet die dominierende Frequenz eines Frames. Allerdings ist diese offensichtlich nicht für jeden Frame bzw. jedes Signal definiert. In manchen Signaltypen gibt es Obertöne und Spitzenwerte (engl. *peaks*), d.h. natürliche Vielfache $k \cdot f_0$ einer Grundfrequenz f_0, die mittels verschiedener Verfahren ermittelt werden können [27].
  Falls vorhanden, kann ggf. die Grundfrequenz als Verteilungsmaximum des als WDF betrachteten Spektrogrammes definiert werden.

- Spektrale Glattheit (engl. *spectal flatness measure*, SFM):
  Dieses Maß stellt eine Charakterisierung zur Unterscheidung von tonalen und rauschartigen Signalen dar. Tonal bezieht sich hier nicht auf den entsprechenden Begriff aus der Musiktheorie, sondern auf das Vorhandensein dominanter sinusartiger Komponenten im Signalspektrum [50]. SFM findet Verwendung im AudioID-System [2] und ist in Form des Low-Level-Deskriptors *AudioSpectrumFlatness* in den MPEG-7-Standard [70] eingeflossen.

- Spectral Flux (auch *Delta Spectrum Magnitude*):
  Dieses Maß charakterisiert framebasierte spektrale Veränderung. Sprachdaten weisen beispielsweise auf Grund der schnellen Abfolge verschiedener Buchstaben sowohl größere als auch variablere Unterschiede auf als Musik. Der Spectral Flux F_t eines gefensterten Signals der Länge N ist definiert als
  $$F_t = \sum_{n=1}^{N} (Spect(t, f_n) - Spect(t-1, f_n))^2 .$$

- Spektrale Roll-Off-Frequenz:
  Dieser Wert misst die Frequenz R_t, unterhalb der sich im Spektrum 95% der Energie befindet und ist somit ein weiteres Maß für spektrale Form. Sie ist definiert durch
  $$\sum_{n=1}^{R_t} Spect(t, f_n) = 0.95 \cdot \sum_{n=1}^{N} Spect(t, f_n) .$$
  Musik weist höhere Energieanteile in hohen Frequenzen auf und besitzt daher eine höhere Rolloff-Frequenz als Sprache.

- 4-Hz Modulationsenergie:
  Sprache besitzt eine charakteristische Energiemodulationsspitze im Bereich der 4-Hz Silbenrate. Um dies zu quantisieren, wird das Signal zur Analyse unter Verwendung des MFCC-Algorithmus' in perzeptuelle Frequenzkanäle konvertiert, mittels eines auf 4Hz zentrierten Filters zweiter Ordnung Bandpass-gefiltert und anschließend zur Erhebung der Lautheit verwendet. Der Maßwert ergibt sich aus der Summe aller normalisierten Kanäle [93].

Zeitbasierte Merkmale:

- Nulldurchgangsrate (engl. „zero-crossing rate"):
  Dieses Audiomerkmal berechnet die Rate der Vorzeichenwechsel der diskreten Abtastwerte eines Frames:
  $$ZCR(t) := \frac{1}{N+1} \sum_{n=1}^{N+1} | \, sign(x'_t[n+1]) - sign(x'_t[n]) \, |$$
  Dabei ist $x'[n]$ der Wert des n-ten Samples. Angenommen wird außerdem, dass x' annähernd mittelwertfrei ist. Hohe ZCR-Werte deuten auf hohe Frequenzen hin.

- Anteil an Low-Energy-Frames:
  Dieses Merkmal beruht auf dem *RMS*-Wert (engl. *root mean square)* des Signals, einem in der Audiosignalverarbeitung üblichen Maß der Amplitude, berechnet durch die Wurzel des quadrierten Signal-Mittelwertes

$$RMS(t) := \sqrt{(\frac{1}{N}\sum_{n=1}^{N} x_t(n))^2} .$$

  Zur Berechnung des Merkmalswertes wird zunächst der Mittelwert aller Frames der letzten Sekunde berechnet. Danach wird für jeden dieser Frames überprüft, ob sein RMS-Wert weniger als 50% des RMS-Durchschnitts beträgt. Zurückgeliefert wird der prozentuale Anteil dieser Low-Energy-Frames an der Gesamtzahl der Frames.

  Durch sprachbedingte leise Frames besitzen Sprachdaten einen üblicherweise höheren Wert dieses Merkmals.

Eine Auflistung weiterer Merkmale findet sich im Anhang der AES'04-Veröffentlichung von Cano et al. [23].

Es sei darauf hingewiesen, dass aufgrund der Ähnlichkeit mancher Audiomerkmale Redundanzen entstehen können. Weiterhin spielen die Form der Fensterfunktion und die Länge der Frames eine wesentliche Rolle bezüglich der Aussagestärke einzelner Audiomerkmale.

e) Nachbearbeitung

Hierunter fallen verschiedene Anpassungen der Merkmalsfolge [21]:

- Ableitung der ermittelten Merkmale [2] bzw. Erzeugung von Bitfolgen aus aufeinanderfolgenden Merkmalsvektoren [57]
- Bildung von Varianz oder Mittelwert der Merkmalsfolge [93]
- Algorithmen zur Verbesserung der Kompaktheit der Merkmale
- Algorithmen zur Verstärkung der Robustheit
- Algorithmen zur Normalisierung
- Vorbereitung auf Hardware-Anforderungen und andere nachfolgende Teilsysteme
- systemspezifische Anpassungen

Die hier vorgestellten Mechanismen des Front-Ends ermöglichen dem im Konzept von Cano et al. [21] nachgeschalteten Fingerabdrucks-Modellierer das Arbeiten auf einer kompakt aufbereiteten Repräsentation der Signaldaten, wie im folgenden Abschnitt beschrieben wird.

2.4.2 Fingerabdrucks-Modellierer

Der Modellierer erhält eine Sequenz von framebasierten Audiomerkmalen und strukturiert diese anhand des zugrunde liegenden Merkmals-Modells. Dieses bildet üblicherweise auch die Basis der nachfolgend verwendeten Module zur Distanz-Metrik und den Indexierungs-Algorithmus.

Meist äußert sich diese Strukturierung in der Umwandlung in eine auch aus Gründen der Speicherplatzeffizienz kompakte Form, beispielsweise als Vektor der multidimensionalen Merkmals-Vektorsequenzen eines gesamten Songs [74] oder als Binärsequenz der Audiomerkmale [45].

Globale Redundanz wird in [2] durch Zusammenfasen (engl. *Clustering*) ähnlicher Merkmale ausgenutzt, wodurch eine Approximation der Vektorsequenz ohne zeitliche Information, ein *Codebuch*, entsteht. Weiterhin werden hier Statistiken über kurze Zeitintervalle hinweg gesammelt, was sich in sowohl höherer Erkennungsrate als auch schnellerem Abgleichen (durch kürzere Sequenzen) widerspiegelt. Eine ausführliche Darstellung dieser Methode wird in Abschnitt 2.5.1 gegeben.

Cano et al. [22] nutzen ein durch Spracherkennung inspiriertes redundanzverringerndes Modell, in dem ähnliche Teilstücke des Audiosignals, wie etwa wiederkehrende perkussive Elemente, als Elemente eines endlichen Alphabets eingestuft werden. Die entstehende Sequenz von Indizes erhält die Information des zeitlichen Audioverlaufes und kann mit geeigneten Mitteln effizient weiterverarbeitet werden. Dieser Ansatz wird in Abschnitt 2.5.3 erläutert.

2.4.3 Distanzmetrik

Ziel der Verwendung einer Distanzmetrik ist das quantitative Einordnen der Ähnlichkeit des Eingangssignals in Bezug auf die in einer Datenbank enthaltenen Audiodokumente. Voraussetzung ist ein metrischer Raum.

Die Wahl des Distanzmaßes hängt auf Grund der Vielzahl von Möglichkeiten eng zusammen mit dem tatsächlich verwendeten Audiofingerabdrucks-Modell. In Abhängigkeit davon wird, wie z.B. in [45] im Fall quantisierter Vektorsequenzen, das Manhattan-Maß oder im binären Fall die Hamming-Distanz angewandt [57]. Weit verbreitet ist auch das Verwenden einer Korrelation zur Messung des Abstandes zweier Signale. Darüber hinaus gibt es eine Reihe von anderen Metriken, die sich jeweils an den speziellen Eigenschaften des gewählten Merkmalsraums orientieren, wie in Abschnitt 2.5 erläutert.

In den Fällen, in denen die zu identifizierenden Signale nicht als Fingerabdruck, sondern als Codebuch [2] vorliegen, wird die aktuelle Merkmals-Sequenz sukzessiv transformiert und mit den einzelnen Codebüchern verglichen, wobei die Fehler je Codebuch akkumuliert werden. Diese gesamte Fehlersumme stellt die Distanz zum Eingabesignal dar.

2.4.4 Suchmethoden

Die Effizienz des Vergleiches eines unbekannten Audiofragmentes mit einer sehr großen Anzahl an Fingerabdrücken wird maßgeblich durch die verwendete Suchmethode bestimmt. Diese wiederum hängt von der Organisation der Fingerabdrücke durch einen Suchindex ab, welcher eine Reduzierung der Anzahl nötiger Vergleiche ermöglicht.

Um den teilweise riesigen Bestand an Daten zu durchforsten, werden approximative Methoden eingesetzt, die stark problemspezifisch sind. Eine gängige Heuristik ist es, durch Nutzen von Äquivalenzklassen viele Trefferkandidaten schnell auszuschließen und nur wenige vollständig zu durchsuchen [25]. Das vorgeschaltete Verwenden eines simpleren Distanzmaßes kann so zur anfänglichen Reduktion möglicher übereinstimmender Datenbank-Dokumente beitragen.

Cano et al. verwenden Stringvergleichs-Methoden aus der Bioinformatik, ähnlich denen zum DNA-Vergleich [22]. In [57] wird ein Index aus Codewörtern verwendet, die aus den Binärsequenzen extrahiert wurden, welche die Merkmalsfolgen repräsentieren.

Da zum einen viele Datenbestände prinzipiell dezentral vorliegen und zum anderen die Hardwareleistungen weiter steigen, stellt sich die Frage der Machbarkeit und Effizienz eines verteilten Audioidentifikationssystems.

In Abhängigkeit des Fingerabdruckmodells und der gegebenen Hardware-Infrastruktur ist es möglich, parallel in mehreren verteilten Datenbanken zu suchen. Kurth et al. [57] beschreiben dazu ein auf Partitionierung der gesamten Datenbank-Dokumentenmenge beruhendes System, dessen einzelne Partitionen unabhängig voneinander durchsucht werden können. Beruhend auf verschiedenen Faktoren einzelner Dokumente und dem Umfang der zu erwartenden Treffermengen lassen sich der zu erwartende Kommunikationsaufwand und der zur Synchronisierung nötige Zeitaufwand einschätzen. Auch die Balancierung der Index-Dokumente ist bei größtenteils ähnlich großen Dokumenten möglich. Die tatsächliche Effizienz hängt aber in großem Maße vom Unfang des Datenbestandes ab, so dass eine Trennung von Indexierung und Suche sinnvoll sein kann.

2.4.5 Hypothesen-Test

Der Verlauf des Vergleichs von Anfrage und Datenbank führt zu einer Menge von Distanzwerten. Um zu entscheiden, ob die Anfrage als ein Dokument der Datenbank identifiziert werden kann, muss der entsprechende berechnete Distanzwert über einem zu definierenden Schwellenwert liegen. Dieser Wert ist stark abhängig vom Fingerabdrucks-Modell, dem Datenbankbestand und dessen Größe und kann bei einer falschen Festsetzung leicht zu „False Positives“ führen, vom System ausgegebenen Treffern, die keine sind. Zur Vermeidung dieser Probleme stehen verfahrensspezifische Fehlertoleranzmechanismen zur Verfügung.

2.4.6 Datenbank-Pflege

Der Vorgang des Einfügens neuer Signal-Dokumente in eine Datenbank folgt grundsätzlich dem Schema, das im Rahmen des Fingerabrucks-Modellierungsvorgangs beschrieben wurde. Zu jedem einzufügenden Signal wird in Abhängigkeit des verwendeten Modells ein Audiofingerabdruck erzeugt und in der Datenbank abgelegt. Damit verknüpft werden zugehörige, evtl. aufzubereitende Metadaten (Typ, Speicherformat, Profildaten, ...), die nach einer Identifikation als Ausgabe zur Verfügung stehen. Falls die zu verwendenden Rohdaten, wie etwa im Falle vieler Audiodateiformate, über eingebettete Strukturdaten verfügen, muss vor dem Extraktionsprozess noch eine Trennung von Audio- und Strukturdaten durchgeführt werden. Ähnliches gilt für komplexe Multimedia-Objekte, die vor der Verarbeitung in einfachere Objekte zerlegt werden müssen. Handelt es sich um besonders lange Signale, kann auch eine Vorsegmentierung nötig sein. Je nach Verfahren kann dieser gesamte Vorgang als Teil der Vorverarbeitungsstufe im Front-End implementiert oder separat vorgeschaltet sein.

Nach dem Einfügen werden ggf. vorhandene Optimierungsverfahren wie Dokumenten-Indexierung oder der Distanzmessung zuarbeitende Vorberechnungen, wie z.B. Einteilung in eine Äquivalenzklasse angewandt, so dass die Datenbank nach Abschluss des Einfügeprozesses wieder auf dem aktuellen Stand ist. Je nach Anforderung muss auch das Löschen und Aktualisieren vorhandener Dokumente möglich sein.

Das Konvertieren bestehender Datenbanken ist nur in Ausnahmefällen möglich, da das Format von Datenbankdokumenten meist so stark anwendungspezifisch ist, dass ein Umwandeln des Dokumentenbestandes nicht möglich ist.

2.5 Aktuelle Audioidentifikationssysteme

Im Folgenden sollen fünf Systeme zur Audioidentifikation erläutert werden, die sich neben guten Erkennungsraten und hoher Robustheit vor allem auch durch Effizienz auszeichnen. Die durchgängig guten Erkennungsraten der einzelnen Systeme werden dabei bewusst nicht zitiert, da zum einen ein objektiver Vergleich aufgrund der Unterschiedlichkeit der verwendeten Audiodaten schwierig ist und zum anderen der Fokus auf den jeweils zugrunde liegenden Prinzipien der einzelnen Systeme liegt.

2.5.1 AudioID

Dieses am Fraunhofer-Institut entwickelte System [2] unterscheidet sich insofern grundlegend von den anderen hier vorgestellten Systemen, als es, einem klassischen Mustererkennungs-Paradigma folgend, klassifikationsbasiert und in zwei Modi arbeitet.

Im Trainingsmodus wird eine Menge von Trainingsaudiosignalen sukzessiv in äquidistanten Abständen anhand von n Frequenzbändern im Spektrum von 300-6000Hz in eine Folge von Merkmalsvektoren im $\mathbb{R}^n$ ungewandelt, wobei Allamanche et al. [2] mit $n = \{4, 16\}$ arbeiteten. Verwendet wurden verschiedene Audiomerkmale. Zum einen gängige Merkmale wie Bandenergie, zum anderen das Spectral Flatness Measure (SFM) und der eng verwandte Spectral Crest Factor (SCF) zur Beschreibung spektraler Glattheit.

Aus diesem Audiofingerabdruck wird mittels eines einfachen k-means-Clustering-Algorithmus, der noch in Abschnitt 2.6 erklärt wird, eine kompakte nicht-temporale Repräsentation des Audiostückes in Form einer kleinen Menge von *Code-Vektoren* im $\mathbb{R}^n$ generiert, das sogenannte *Codebuch*. Die Anzahl dieser Vektoren ist abhängig von einem zu definierenden maximalen *RMSE* (root mean square error) und kann je nach Datenbeschaffenheit gut abgeschätzt werden. Jedes Codebuch stellt eine Klasse dar und wird entsprechend in einer Klassendatenbank abgelegt.

Die im Erkennungsmodus stattfindende Identifikation erhält eine Anfrage in Form einer Folge von Vektoren im $\mathbb{R}^n$. Die Treffererkennung stellt nun ein N-Klassen Klassifikationsproblem dar: Zunächst wird die Anfrage in eine Sequenz entsprechender Merkmale transformiert, die nachfolgend durch alle in der Datenbank vorliegenden Codebücher unter Verwendung der RMSE-Distanz approximiert wird. Dabei wird der Fehler jedes Codebuches akkumuliert. Die Anfrage wird derjenigen Klasse zugeordnet, die den geringsten akkumulierten Fehler besitzt. Zur Steigerung der Performanz werden während der Verarbeitung der Vektorfolgen kurzzeitliche Statistiken berechnet und im folgenden angewandt, was zu einer sowohl höheren Erkennungsrate wie auch zu einer schnelleren Berechnung führt. In Tests mit bis zu 15.000 Testmusikstücken erreichte das System selbst bei verrauschten Signalen über 98% korrekter Treffer. Darüber hinaus ist AudioID ein sehr performantes System, das auch auf gängigen Desktop-PC-Systemen läuft.

2.5.2 Shazam Ent. / Wang

Als eines der ersten Verfahren wird der von Shazam Entertainment (Kalifornien) unter Leitung von Avery Wang entwickelte Audioidentifikationsalgorithmus [114] seit 2002 kommerziell eingesetzt. Der unter anderem in Großbritannien und in Deutschland von Vodafone angebotene Service erlaubt eine Identifikation von per Mobiltelefon übertragenen Musikstücken. Der kontinuierlich steigende Datenbestand umfasst nach eigenen Angaben derzeit über 2,2 Millionen Stücke. Zur Identifikation werden die ersten 30 Sekunden des Service-Anrufes mitgeschnitten und auf dem firmeneigenen Cluster aus 80 vernetzten Linux-PCs analysiert [104].

Der Algorithmus arbeitet ereignisbasiert mit Audio-Merkmalen , die aus Energiemaxima, also Spitzen im Spektrogramm des Signals, gewonnen werden. Diese Merkmale beschreiben Ankerpunkte in einer Zeit-Frequenz-Konstellationskarte, die mittels einer durch Unterschiede in Frequenz und Zeit definierten Nachbarschaftsrelation zueinander in Beziehung gesetzt werden. Jedem Ankerpunkt wird somit eine geeignete Zone im angrenzenden Bereich der Konstellationskarte zugeordnet. Die Merkmale werden nachfolgend klassifiziert und in einer durch die Merkmalsklasse indexierten Hash-Tabelle gespeichert [57].

Die Treffersuche verwendet ein Votingverfahren, das mögliche Verschiebungen zwischen Anfrage und Datenbestand berechnet und sich durch seine Anlehnung an *Geometric Hashing* [118] sehr gut parallelisieren lässt. Dabei wird ein Ranking vorgenommen, das auf einem aus der Differenz fortlaufender Offsets zwischen Anfrage und Datenbasis berechneten Histogramm beruht. Der Ranking-Wert ergibt sich aus der Höhe der größten Spitze des Histogramms.

Wang et al. beschreiben in [114] die Erkennungsleistung des Systems von Audiostücken verschiedener Länge. Dabei fällt auf, dass die prozentualen Erkennungsraten von GSM-kodierten (*Global Standard for Mobile Communications*, Mobilfunk-Protokoll) Daten denen von unkodierten PCM-Daten mit nur 6dB schlechterem Signal-zu-Rausch-Abstand stark ähneln. Neben seiner als effizient beschriebenen Eigenschaft kann dem Algorithmus daher wohl eine gute Robustheit im Umgang mit im Mobilfunksegment typischen Signalen attestiert werden.

2.5.3 AudioDNA

Das von Cano et al. entwickelte AudioDNA-System [22] verwendet zum einen Methoden aus der Bioinformatik, zum anderen die in der Spracherkennung erfolgreich eingesetzten *Hidden Markov Models* (HMM). Diese stellen im vorliegenden Kontext abstrakte Generatoren in Form endlicher Automaten dar, die zustandsbasiert eine reproduzierbare Folge von Symbolen eines Alphabets ausgeben.

Die Umwandlung des Signals in eine solche Symbolfolge findet in mehreren Schritten statt. Zunächst werden MFCCs aus dem Signalspektrum erzeugt, die zur Komprimierung der niederwertigeren Koeffizienten und deren Dekorrelation nachfolgend einer Diskreten Kosinustransformation (DCT) unterzogen werden. Um die als Voraussetzung nachfolgender Schritte nötige Unabhängigkeit der Merkmalsvektoren zu ihren Nachbarn näherungsweise zu garantieren, werden auch die ersten beiden Ableitungen hinzugezogen.

Unter Verwendung akustischer Modelle wird nun zu der gegebenen Vektorsequenz eine Anzahl von HMMs generiert, die als *AudioGenes* bezeichnet werden und gewissermaßen Generatoren für die übergebenen Signalfolgen darstellen. Diese AudioGenes, in die Angaben über zeitliche Positionen eingebettet sind, bilden zusammen die sogenannte *AudioDNA*. Die Dekodierung der Merkmalsfolge, also die Berechnung der wahrscheinlichsten AudioDNA, wird dabei durch den effizienten Viterbi-Algorithmus [112] durchgeführt. Dieser findet die wahrscheinlichste Zustandsfolge einer durch die verwendeten HMMs emittierten Symbolfolge. Die AudioDNA wird nachfolgend in einer Baumstruktur abgelegt.

Das zur Identifikation übergebene Audiosignal wird ebenfalls in AudioDNA umgewandelt und per aus der Bioinformatik bekanntem schnellem approximativem Stringmatching mit den im Suchbaum vorhandenen AudioDNA-Elementen verglichen. Die eigentliche Identifikation erfolgt durch Kombination der verschiedenen AudioGenes und der zugehörigen Positionsangaben.

Cano et al. legen in ihrer Arbeit besonderen Wert auf Robustheit bei Verwendung von Radiosignalen, insbesondere auf die Kompensation zeitlicher Skalierungen gesendeter Beiträge, was zur Verwendung von Hidden Markov Modellen führte.

Neben Erweiterungen des ursprünglichen Konzeptes entstand aus dem Kontext des AudioDNA-Verfahrens das gut skalierbare Amadeus-System [14], welches ein auch flexibel auf andere Bereiche des Multimedia Information Retrievals adaptierbares Identifikations-Instrument darstellt.

2.5.4 AudioHashing

J. Haitsma und T. Kalker von Philips Research extrahieren in ihrem von Methoden aus der Kryptografie inspirierten Ansatz [45] in regelmäßigen Abständen von 11,8 ms Fingerabdrücke aus dem Signalfluss. Dabei genügt nach ihren Angaben ein Audiosignal mit einer gesamten Länge von nur 3 Sekunden für eine Identifikation.

Nach dem Downsampling auf eine Abtastrate von 5 kHz wird das per STFT erzeugte Spektrum auf 300 – 2000 Hz eingegrenzt und in 33 logarithmisch gestufte Frequenzbänder eingeteilt. Für jedes Band wird die Bandenergie berechnet und aus der als ein Bit interpretierten Differenz zweier benachbarter Bänder ein 32 Bit langer, *Hash-Signatur* genannter Wert erzeugt. Eine Menge von 256 solcher Signaturen werden zu einem *Hash-Block* zusammengefasst und repräsentieren somit in 8192 Bit ca. 3 Sekunden des Audiosignals. Die Wahrscheinlichkeit des gleichen Auftretens zweier aus unterschiedlichen Audiosignalen erzeugten Hash-Blöcke $A, B \in \{0,1\}^{32 \cdot 256}$ ist sehr gering, ähnlich wie bei Zahlen, die durch eine geeignete Hash-Funktion generiert werden. Zur Indexierung eines Hash-Blockes werden alle 256 Signaturen in einer Hash-Tabelle abgelegt, zusammen mit einer Referenz auf den nachfolgenden Hash-Block.

Zwei Hash-Blöcke gelten dann als *ähnlich*, wenn ihr Hamming-Abstand (hier in Form der Bitunterschiede) kleiner einem gewissen Grenzwert ist. In den publizierten Experimenten zeigte sich, dass bis zu 2867 der 8192 Bits eines Blocks Fehler sein dürfen, um eine Identifikation noch zu ermöglichen.

Eine zu identifizierendes Audiosignal wird ebenfalls in eine Folge von Hash-Signaturen umgewandelt. Da sich ein Brute-Force-Vergleich aus Performanzgründen verbietet, werden unter der Annahme, dass mit hoher Wahrscheinlichkeit auch ein verzerrtes Audiosignal zu einem dem Original ähnlichen Hash-Block führt und dabei mindestens eine Signatur je Block fehlerfrei bleibt, nur bestimmte Bit-Positionen verglichen. Bei Übereinstimmung wird der weitere Hash-Block genauer untersucht. Dieses Vorgehen spiegelt sich in geringer Laufzeit und hohen Trefferquoten wider. Da in der Praxis die angenommene Robustheit bei stark verzerrten Signalen nicht immer gewährleistet ist, erlaubt man zur Steigerung der Robustheit 1-2 Bitfehler. Dies führt allerdings zu einer höheren Laufzeit.

Das rechenintensivste Teilmodul ist jedoch die Fouriertransformation, die das Fingerabdrucksystem jedoch in Form einer reellwertigen Festkomma-Implementierung selbst auf einem PDA oder Mobiltelefon lauffähig macht.

Die von Philips kommerziell vertriebene Implementierung besteht aus einer einfachen Master/Slave-Architektur, die sich gut skalieren lässt. Weiterhin ist der eigentlichen Identifikation eine unabhängige Klassifikation mit Qualitätseinschätzung vorgeschaltet.

2.5.5 Audentify!

Das in der Arbeitsgruppe Multimedia-Signalverarbeitung der Universität Bonn von Professor Clausen unter Mitarbeit von F. Kurth und A. Ribbrock entwickelte Konzept zur Audioidentifikation [57, 58, 13, 91] basiert auf einem gruppentheoretischen, Operator-basierten Ansatz. Aufsetzend auf einer allgemeingültigen Theorie zur Konstellationssuche mit G-invertierten Listen [57] wurde diese für die Arbeit mit Audiodaten hin konkretisiert. Die entwickelte Suchtechnik umfasst sowohl Translations-, wie in bestimmten Varianten auch Skalierungsinvarianz und verwendet eine klare Trennung von Merkmals-Extraktion und Such- bzw. Indizierungstechnik.

Zur flexiblen Handhabung von verschiedenartigen Systemanforderungen wurden mehrere Merkmals-Extraktoren entwickelt, die jedoch darin übereinstimmen, dass sie Signale auf *k-signifikante Maxima* untersuchen, Positionen im Signal, deren Amplitude höher ist als die der *k* nächsten Nachbarn, und anhand der Merkmalsfolgen in Klassen einteilen. So untersucht der F_{Vol}-Extraktor [91] die Abstände benachbarter aus der Lautheit des Signals extrahierter *k*-signifikante

Maxima, der F_{WFT}-Extraktor hingegen die der Maxima der Mittelfrequenz-Folge des Signal-Spektrums.

Einen gänzlich anderen Ansatz bieten auf der Codierungstheorie aufbauende Extraktoren, die der oft ungünstigen Merkmalsverteilung entgegen zu wirken versuchen. Diese basieren auf Merkmalen wie etwa der Lautheit eines Signals und ermöglichen die Umwandlung der resultierenden Merkmalsfolge mittels eines binären Quantisierers und eines linearen, fehlerkorrigierenden Codes in binäre Codewörter $c \in \{0,1\}^n$ gleicher Länge. Anhand dieser Codewörter werden die Signale in Klassen eingeteilt. Die gewählten Vektorräume der Codewörter streben dabei einen möglichst großen paarweisen Hamming-Abstand an. Zur Erzeugung der Codes werden unter anderem approximative Lernverfahren eingesetzt. Eine Erweiterung stellen Codewörter der Form $(0^+1^+)^+$ dar [13], die durch nicht-uniforme Länge ein Arbeiten auf skalierten Signalen erlauben.

Der zugrunde liegenden Theorie folgend werden die Dokument, Translationswert und ggf. Skalierungsfaktor umfassenden Dateneinträge in invertierten Listen abgelegt. Zur Trefferbildung, die zu einer Menge von Treffern führt, die sich durch Dokumentenindex und Translationsinformation auszeichnen, wird die Anfrage sukzessiv per Schnittmengenbildung mit den invertierten Listen verknüpft. Entscheidend für die Effizienz ist daher die Kürze der einzelnen Listen. In der Praxis hat sich zu diesem Zweck die dynamische Speicherung in AVL-Bäumen bewährt, die ein schnelles Finden mittels binärer Suche erlauben.
Erweiterungen dieses Schemas ergeben sich aus fehlertoleranter und unscharfer Suche, sowie aus einem Ranking einzelner Trefferkandidaten. Anwendung findet das Konzept unter anderem im Sentinel-System [58], das in Kapitel 2.7 erläutert wird, und im Rahmen des in dieser Diplomarbeit entworfenen generischen Monitoringsystems, das in den Kapiteln 4 und 5 vorgestellt wird.

Für eine Übersicht über weitere Ansätze zur Audioidentifikation sei hier auf die Arbeit [21] verwiesen.

Über die reinen Audioidentifikations-Technologien hinaus kam es in den letzten Jahren zu einem Boom von Start-Up-Unternehmen, deren Produkte sich mit Audioidentifikation beschäftigen, darunter das von Napster-Gründer Shawn Fanning gegründete Relatable [87]. Aufgrund rechtlicher Schwierigkeiten der Firma Napster war die neueste Version des Systems, die eine High-Level-Musikmodellierung zur Filterung von File-Sharing-Anfragen verwendete, jedoch nur wenige Tage im Einsatz. Die Technologie wurde von Fanning im Dezember 2004 in einem neuen Service namens Snocap [98] der Öffentlichkeit vorgestellt. Weitere Firmen und Projekte finden sich online, z.B. AudibleMagic [9], Yacast [120] und MusicBrainz [74].

2.6 Audioklassifikation

Ein der Audioidentifikation nah verwandter Teilbereich des *Audio Information Retrieval* ist die *Audioklassifikation*. Ziel ist dabei die automatische Zuordnung eines Anfrage-Audiosignals zu einer semantischen Klasse. Auch hier unterscheidet man auf Metadaten arbeitende Verfahren, z.B. Stringmatching unter Verwendung von Anmerkungen zu Autor und Titel, und inhaltsbasierte Verfahren. Hier soll auf letzteren Fall eingegangen werden. Einige der hierbei erläuterten Konzepte finden Anwendung in dem Klassifikationsmodul, welches im Rahmen dieser Diplomarbeit entwickelt und umgesetzt wurde. Die entsprechende Darstellung findet sich in Kapitel 5.3.10.

Übliche Audioklassifikationsszenarien haben eine Einordnung eingehender Signaldaten als Musik oder Sprache zur Aufgabe [93], ebenso wie die Klassifikation von Musik in verschiedene Genres [108] oder die Klassifikation von in Audioaufnahmen vorhandenen Musikinstrumenten [47]. Reale Anwendungen ergeben sich z.B. in der Medizin, wo ein Expertensystem

Atmungsgeräusche analysiert, oder im Bereich der Heimelektronik, etwa zur automatischen Auswahl eines entsprechenden Equalizer-Presets anhand des erkannten Musikgenres [42].

Die Verwendung von Audioklassifikatoren stellt eines der Hauptelemente von Audiomonitoringsystemen dar. Dies reicht von Vorverarbeitungsschritten zur Effizienzsteigerung oder zur automatischen Segmentierung von Audioströmen [8], etwa um Datenströme automatisch zu beschriften [107], bis hin zur automatischen Diskriminierung von Nachrichtensendungen und Werbung in Datenströmen [62].

Auch andere Bereiche profitieren von effizienten Audioklassifikations-Verfahren. Da Musik und Sprache durch unterschiedliche Komprimierungsverfahren (Codecs) jeweils effizient bearbeitet werden können, universelle Codecs im Allgemeinen jedoch weniger Codierungseffizienz in Form geringerer Komprimierungsraten besitzen, könnte durch Verwendung eines Moduls zur Klassifikation in Musik und Sprache eine flexible, effektivere Komprimierung eingesetzt werden [34], etwa für Übertragung von Multimediadaten im Internet. Weitere denkbare Einsatzgebiete finden sich im Umgang mit Audiobibliotheken, die durch automatische Klassifikation von Audiosignaldaten gegenüber menschlichen Einordnungen sowohl von einer objektiveren Indexierung als auch von einer Geschwindigkeitssteigerung profitierten.

Die zur Klassifikation verwendeten Methoden entstammen meist der Mustererkennung und lassen sich grundsätzlich in überwachte und unüberwachte Verfahren unterteilen [52]. Überwachte Klassifikation beschreibt die Zuordnung eines Objektes zu einer von mehreren vordefinierten Klassen, während die zu Beginn leeren Klassen im unüberwachten Fall selbstständig erlernt werden. Durch ihre Verwandtschaft teilen Audioklassifikationsverfahren viele Eigenschaften mit Methoden aus anderen wissenschaftlichen Gebieten, wie z.B. der Biometrik, Data Mining und der Bildanalyse.

Audiosignale werden üblicherweise als *Muster* durch möglichst robuste *d*-dimensionale Merkmale repräsentiert und erlauben so das Verwenden der auch in der Audioidentifikation eingesetzten und in Abschnitt 2.3 und 2.4 erläuterten Verfahren. Die enge Verwandtschaft zu diesen Verfahren zeigt sich auch im grundsätzlich ähnlichen Aufbau, der aus Vorverarbeitung, Merkmalsextraktion und Entscheidungsmodul besteht. Entscheidend für die Effektivität des dabei verwendeten Merkmalsraumes ist, wie gut Muster verschiedener Klassen voneinander separiert werden können und somit die Qualität der die Klassen abgrenzenden *Entscheidungsgrenzen* (engl. *decision boundaries*). McKinney et al. [65] zeigen in ihrer Arbeit, dass die Wahl speziell auf das konkrete Problem zugeschnittener Merkmale (z.B. aus der Psychoakustik, im Gegensatz zu üblich verwendeten Merkmalen) einen großen Einfluss auf die Effektivität der Klassifikation besitzt.

Scheirer und Slaney verwenden in ihrer vielzitierten Veröffentlichung [93] zur Diskriminierung von Sprache und Musik dreizehn verschiedene Merkmale und untersuchen Effektivität und Effizienz einzelner Merkmale. Neben typischen hier in Abschnitt 2.4.1d genannten Merkmalen in der Spracherkennung und deren Varianten, die teilweise bessere Ergebnisse erzielten als die zugrunde liegenden Merkmale, präsentieren die Autoren auch neue Merkmale, darunter das „Pulse-Metric"-Merkmal, das Rhythmizität anhand von Autokorrelationen über 5 Sekunden lange Zeiträume analysiert und das zu den drei effektivsten der untersuchten Merkmale gehört. Weiterhin werden zur Untersuchung der Praxistauglichkeit die Klassifikationsfehler der besten und der schnellsten Verfahren gegenübergestellt.

Neben dem weit verbreiteten *statistischen* Ansatz spielen vor allem von biologischen Mustererkennungsverfahren inspirierte *Neuronale Netzwerke* eine wichtige Rolle. Diese besitzen eine integrierte Einheit aus Merkmalsextraktion und Klassifikator und arbeiten durch verschaltete Schichten bzw. Netze einzelner trainierbarer Module, den *Neuronen*. Die Topologie und das Zusammenspiel einzelner Neurone variiert je nach Netztyp und umfasst auch rückgekoppelte Netze und selbstorganisierende bzw. topologiebewahrende Karten [121]. Die Anzahl der

Neurone je Schicht korrespondiert dabei mit der jeweiligen Dimension der verarbeiteten Daten. So lässt sich beispielsweise in einem Multilayer-Perzeptron mittels zwischen Ein- und Ausgabeschicht befindlichen *versteckten* Schichten eine je nach Typ lineare oder nichtlineare Repräsentation der Hauptkomponenten der Daten erreichen [52].

Überwachte statistische Klassifikationsverfahren arbeiten in zwei Modi: Im Trainingsmodus wird der Klassifikator trainiert, den Merkmalsraum mittels einer Menge von Trainingsmustern zu partitionieren. Anhand von Feedback-Mechanismen können dabei Vorverarbeitungsparameter als auch die Merkmalsextraktion optimiert werden.

Im Klassifikationsmodus wird ein durch einen *d*-dimensionalen Merkmalsvektor repräsentiertes Muster anhand einer Entscheidungsregel einer Klasse zugeordnet. Da die Verteilungswahrscheinlichkeiten der Klassen in der Praxis üblicherweise unbekannt sind und im Trainingsmodus abgeschätzt werden, finden aus der Mustererkennung bekannte optimale Verfahren wie die Bayes-Regel meist keine direkte Anwendung. Falls die Form der Verteilung bekannt ist (z.B. Gaußsch), mehrere Parameter jedoch unbekannt, spricht man von einem „parametrischen Entscheidungsproblem". Eine in diesem Fall übliche Herangehensweise stellt das Ersetzen der unbekannten Werte durch Schätzungen dar. Falls keinerlei Informationen über die Wahrscheinlichkeitsverteilung der Klassen vorliegen, spricht man dagegen von einem „nicht-parametrischen Entscheidungsproblem", in dem auf den Trainingsdaten basierte Entscheidungs-grenzen konstruiert werden müssen, z.B. über die *k-nearest-neighbour*-Regel oder die Verwendung eines Multilayer-Perzeptrons.

Ein in der Praxis entscheidender Faktor betrifft die Auswahl der Trainingsdaten. Durch unzureichendes Wissen über zu klassifizierende Audiosignale und der daraus meist resultierenden Abhängigkeit von Schätzungen müssen die im Trainingsmodus verwendeten Trainingsdaten besonderen Ansprüchen genügen: Zum einen muss ihre Anzahl angemessen sein, zum anderen sollten ihre Werte typisch für nachfolgende zu klassifizierende Werte sein. Bei Verwendung zu weniger Trainingsmuster kann der Klassifikator den Merkmalsraum nicht gut modellieren, zu viele Daten können hingegen zum Effekt des *Übertrainierens* führen, bei dem sich der Klassifikator zu sehr den Parametern der Trainingsdaten angepasst hat. Strategien zur Vermeidung solcher Probleme weisen auf rotierende Trainingsmengen hin, die unabhängig zu Optimierung und Evaluation eingesetzt werden [52]. So verwenden Scheirer und Slaney [93] für ihren Klassifikator zur Diskriminierung von Musik und Sprache je 20 Minuten an Audiodaten, die sie aus Radioaufnahmen gewinnen. Dabei legen sie Wert auf eine repräsentative Auswahl verschiedener Genres mit und ohne Gesang, sowie Stimmen beider Geschlechter mit verschiedenen Hintergrundgeräuschen und typischen Verzerrungseffekten.

Die Trainingsdaten geben darüber hinaus Auskunft über die Fehlerrate des Klassifikators, d.h. den Prozentsatz falsch klassifizierter Daten. Wenn die Menge der Trainingsdaten groß und aussagekräftig genug ist und Trainings- und Klassifikationsmuster unabhängig sind, gibt dieser Wert eine gute Schätzung der tatsächlichen Fehlerrate. Diese ist dagegen selbst für den Fall, dass sämtliche System-Parameter vorliegen, äußerst schwierig analytisch festzusetzen und ist in der Praxis in einem solchen Fall bestenfalls per Bayes'schem Fehlermaß abzuschätzen. Für eine Übersicht weiterer Fehlermaße siehe Jain et al. [52].

Die Performanz eines Klassifikators hängt darüber hinaus auch vom Verhältnis aus Datengröße, Merkmalszahl und Klassifikator-Komplexität ab. Bei Einteilung der Trainingsdaten in Klassen steigt die Zahl der Trainingspunkte im Merkmalsraum exponentiell mit der Größe der Merkmalsdimension – ein „Fluch der Dimensionalität" getauftes Problem, das auch in vielen anderen Wissenschaften die Arbeit mit großen Datenmengen beschränkt. Jain et al. [52] weisen in diesem Zusammenhang darauf hin, dass durch ein Verhältnis von mindestens zehn mal so vielen Trainingsmustern per Klasse wie der Anzahl verwendeter Merkmale eine Vermeidung des Problems erreicht werden kann.

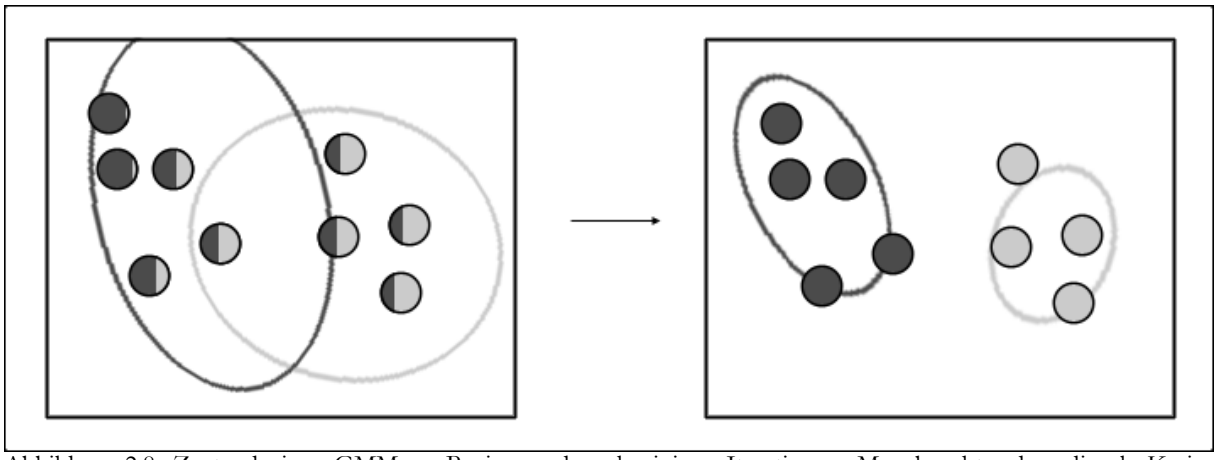


Abbildung 2.8: Zustand eines GMM zu Beginn und nach einigen Iterationen. Man beachte, dass die als Kreise dargestellten Merkmalsvektoren eine sich im Laufe der Iterationen verändernde Wahrscheinlichkeit der Klassenzugehörigkeit aufzeigen.

Klassifikatoren lassen sich je nach enthaltenem Entscheidungsverfahren in verschiedene Typen einteilen. Der einfachste Ansatz verwendet sogenanntes *Ähnlichkeitsmaß*, um Muster Klassen zuzuordnen. Dazu wird das Anfrage-Muster mit Repräsentanten jeder Klasse verglichen und anhand des Ähnlichkeitsmaßes der naheliegendsten Klasse zugeordnet. Die Berechnung der Repräsentanten kann per Mittelwert der Klasse (*nearest mean classifier*) oder auch durch komplexere Verfahren wie Vektor-Quantisierung (VQ) erfolgen. Ein typischer solcher Klassifikator ist der *k-nearest neighbour*-Klassifikator (k-NN), bei dem per Euklidischer oder Mahalanobis-Distanz zu einem eingehenden Punkt im Merkmalsraum die *k* nächstliegenden Punkte berechnet werden. Die Zuordnung erfolgt dann zu derjenigen Klasse, der die Mehrheit dieser *k* Punkte zugeordnet ist. Nachteil ist die nötige Speicherung einer großen Zahl von Trainingsvektoren und die dadurch hohe Anzahl an Vergleichen. Zur Effizienzsteigerung werden daher unter anderem räumliche Partitionierungsstrategien wie *k-d*-Bäume eingesetzt [93], die den Merkmalsraum per Manhattan-Maß vorpartitionieren und so die Anzahl nötiger Vergleiche drastisch reduzieren.

Ein anderer Klassifikatoransatz verwendet Wahrscheinlichkeitsberechnungen und parametrische Modelle, um über Schätzungen von Parametern eine Entscheidung über eine Klassenzugehörigkeit eines zu klassifizierenden Objekts durch Wahl der höchsten Wahrscheinlichkeit zu ermöglichen. Ein typischer solcher Klassifikator ist z.B. der mehrdimensionale Gaußsche MAP (*maximum a posteriori*)-Schätzer, der jede Klasse als Gaußsche Verteilung von Punkten eines multidimensionalen Raumes interpretiert. Anhand von durch Trainingsdaten geschätzten Mittelwert- und Kovarianz-Parametern jeder Klasse können eingehende Anfragemuster anhand der Mahalanobis-Distanzmessung der nächsten Klasse zugeordnet werden [93]. In ähnlicher Weise modellieren Gaußsche Mixtur-Modelle (GMM) Klassen als Vereinigung mehrerer Gaußscher Cluster. Siehe auch Abbildung 2.8. Anhand geschätzter Wahrscheinlichkeitsmodelle werden Anfragen der wahrscheinlichsten Klasse zugeordnet. Die Schätzungen werden dabei iterativ durch den Expectation-Maximization-Algorithmus durchgeführt, wie im Fall des von Vacher et al. [110] zur Sprachklassifikation im Rahmen eines medizinischen Telesystems verwendeten GMMs.

Eine direkte (*geometrische*) Berechnung der Entscheidungsgrenzen kann jedoch auch zu einem dritten Klassifikatoransatz führen. Dazu wird ein Fehlerkriterium optimiert, beispielsweise der durchschnittliche quadrierte Fehler (engl. *mean squared error*, *MSE*). Dies kann beispielsweise durch Neuronale Netze geleistet werden [52].

Einen relativ neuen Ansatz findet man im Bereich der *Support Vector Machines*. Diese zwei-Klassen-Klassifikatoren berechnen eine Menge von Vektoren, die den (geometrisch interpretierten) leeren Randbereich zwischen den beiden zu trennenden Klassen definieren [52].

Auch die Verwendung der in der Spracherkennung erfolgreich eingesetzten *Hidden Markov Models* (HMM) bietet, insbesondere in Verbindung mit anderen Verfahren, neue Möglichkeiten, vor allem im Bereich der Audio-Segmentierung [56]. Zhang et al. setzen HMMs in ihrem hierarchischen Klassifikationssystem ein, um regelbasierte Grobklassifikationen von Audiosignalen weiter zu verfeinern [122].

Da jedes Klassifikationsverfahren problemspezifische Vor- und Nachteile besitzt, kann durch eine Verschaltung mehrerer Klassifikatoren eine Effektivitätssteigerung erzielt werden. Die Möglichkeiten umfassen verschiedene Architekturen und Gewichtungen. So kann etwa durch eine serielle Kombination schneller, aber ungenauer Klassifikatoren und langsamer, effektiver Klassifikationsverfahren eine Leistungsverbesserung erzielt werden. Für eine Übersicht siehe Jain et al [52]. Eine ähnliche Möglichkeit stellt eine dynamische Gewichtung, z.B. in Form eines gleitenden Mittelwertes dar, mittels dem eine fehlerhafte Klassenzuordnung (engl. *class switching*) eingeschränkt werden kann [34].

Eine unüberwachte Audioklassifikation muss im Gegensatz zu Ansätzen, die auf vorklassifizierten Trainingsdaten arbeiten, selbstständig semantische Klassen generieren. Dies kann durch eine regelbasierte Heuristik geschehen, wie von Djeraba et al. [31] beschrieben wird. Die Autoren skizzieren ein System zur Klassifikation von Audiosignalen in Stille, Rauschen, Sprache und Musik, wobei die Klassifikation nach Anwenden einer Fensterfunktion wie folgt durchgeführt wird: Zunächst wird versucht, das Signal als Stille über einen *Selbst-Normalisierung* genannten adaptiven Ansatz zu erkennen, der nach der Erkennung die eigenen Parameter aktualisiert. Die Autoren weisen darauf hin, dass eine Erkennung mittels der Lautheit des Signals wegen in der Praxis zu erwartender Hintergrundgeräusche nicht ausreiche. Mittels einer Autokorrelationsfunktion wird nachfolgend die Periodizität gemessen. Falls keine Periodizität vorliegt, wird die Anfrage als Rauschen klassifiziert. Falls kein Rauschen vorliegt, wird die Nulldurchgangsrate des Signals untersucht. Ist diese höher als ein Grenzwert, wird die Anfrage als Sprache, im anderen Fall als Musik klassifiziert. Panagiotakis et al. verwenden serielle Tests aus Kombinationen einfacher Merkmale zur Diskriminierung von Sprache und Musik und setzen eine vorgeschaltete Erkennung der Signalpräsenz anhand eines Maßes der Signalamplitude zur Klassifikation von Stille ein. Der Vorteil solcher Verfahren liegt in ihrer Effizienz [82].

Liegt ein solches Regelsystem nicht vor, kann ein *Clustering-Algorithmus* zu Hilfe gezogen werden, der unter Verwendung eines Cluster-Kriteriums Muster in selbstgenerierte Klassen (*Cluster*) einteilt. Neben hierarchischen Methoden [52], auf die hier nicht weiter eingegangen werden soll, gibt es Techniken, die auf Partitionierungstrategien beruhen und sich üblicherweise eines quadratischen Fehler-Kriteriums bedienen. Ziel ist dabei die Generierung einer Partition aus einer fixen Anzahl von disjunkten Clustern, wobei ein quadratisches Kostenmaß minimiert werden soll. Jedes Cluster wird durch seinen Mittelpunkt, den *Centroid*, repräsentiert, der nach Einfügen neuer Muster aktualisiert wird. Der quadratische Fehler jedes Clusters ist die Summe der Euklidischen Entfernungen jedes Musters zum Centroid. Ziel ist die Minimierung der Summe aller Cluster-Fehler. Der auf diesem Ansatz basierende *k-means*-Algorithmus ist effizient und liefert unter bestimmten Umständen erstaunlich gute Ergebnisse [69, 52]. Leong setzt ihn beispielsweise im Rahmen eines Musikidentifzierungssystems ein, um MFCC-Merkmale zu clustern [59].

2.7 Audiomonitoringsysteme

Der Begriff des Audiomonitorings beschreibt diejenige Spezialisierung des in Abschnitt 2.1 beschriebenen allgemeinen Multimediamonitorings, die auf den erläuterten Komponenten Audioidentifikation und/oder -Klassifikation aufbaut. Die in der Praxis verwendeten Systeme bedienen sich dabei der gesamten Spannweite der in den vorigen Abschnitten erläuterten Ansätze. Oft finden sich dabei Kooperationen zwischen Forschungsgruppen und kommerziellen Interessenten, wie unter dem Punkt *Monitoring am Übertragungskanal* nachfolgend beschrieben wird. Wie in Abschnitt 2.1 erläutert, ist ein zentrales Merkmal dieser Systeme ihre Echtzeit-Fähigkeit. Da es wegen der Vielfältigkeit der Zielsetzungen und den verwendeten Frameworks kaum möglich scheint, eine allumfassende Definition des Audiomonitorings zu geben, soll im Folgenden zunächst auf übliche Gemeinsamkeiten vieler Systeme eingegangen und danach ein konkreter Überblick über ausgewählte Beispiel-Anwendungen gegeben werden.

Nach Cano et al. [21] lassen sich Audiomonitoring-Systeme je nach Anwendungsbereich in drei Klassen einteilen:

- **Monitoring auf Seiten des Distributors**
  Anbieter von Inhalten können feststellen, ob sie über die zur Veröffentlichung nötigen Berechtigungen verfügen. Auch die Arbeit mit archiviertem Audiomaterial kann durch Monitoring profitieren. Das gilt auch für Anbieter von CD-Verfältigungs-Dienstleistungen, die auf diese Weise nicht autorisierte Inhalte überprüfen können [90, 10].

- **Monitoring am Übertragungskanal**
  Sowohl Rechteinhaber von z.B. im Radio übertragenen Musikstücken als auch gewerbliche Werbekunden haben ein Interesse daran, gesendete Inhalte zu überprüfen – sei es, um Übertragungsentgelte zu überwachen, tatsächliches *Airplay* von Werbeanzeigen zu kontrollieren oder aus statistischen Gründen. Dazu zählen neben der Generierung von Spiellisten und der Erstellung von Charts auch die Kontrolle staatlicher Vorgaben, etwa in Form von Quoten, wie sie beispielsweise in Frankreich vorgeschrieben sind und derzeit auch in Deutschland diskutiert werden [99]. Anbieter solcher Audiomonitoring-Dienstleistungen sind Nielsen Broadcast Data System [79] und MusicTrace [76], die das am Fraunhofer-Institut entwickelte AudioID-Verfahren verwenden, das in Abschnitt 2.5.1 beschrieben wurde. Weitere Firmen in diesem Segement sind Music Reporter [73], deren Technologie auf dem in Abschnitt 2.5.3 beschriebenen AudioDNA-Verfahren aufsetzt [22] und Audible Magic [9]. Letztere kauften im Jahr 2000 Muscle Fish, eine Firma, die eines der ersten Audioklassifikationssysteme entwickelte [117].
  Weitere Anwendungen ergeben sich aus Filesharing-Applikationen. Siehe dazu beispielsweise die in Abschnitt 2.5 angeführten Bemerkungen zu Relatable und Napster. Auch auf Webseiten befindliche Inhalte können mittels geeigneter Software analysiert und dokumentiert werden, etwa die der Firma Bay TSP [15].

- **Monitoring beim Konsumenten**
  Bei privaten wie gewerblichen Kunden eingesetzte Audiomonitoring-Anwendungen können vor allem dazu dienen, digitales Rechte-Management (DRM) durchzusetzen, etwa durch einen Echtzeit-Abgleich mit online befindlichen Datenbanken und einer bei entsprechender Verletzung von Rechten automatischen Abspielsperre von Geräten und/oder Software. Hier spielen auch digitale und/oder akustische Wasserzeichen eine wichtige Rolle.
  Darüber hinaus können sich auch hier aus Filesharing-Netzwerken neue Anwendungen ergeben, etwa zur Verifizierung der Echtheit und Vollständigkeit von Musikstücken.

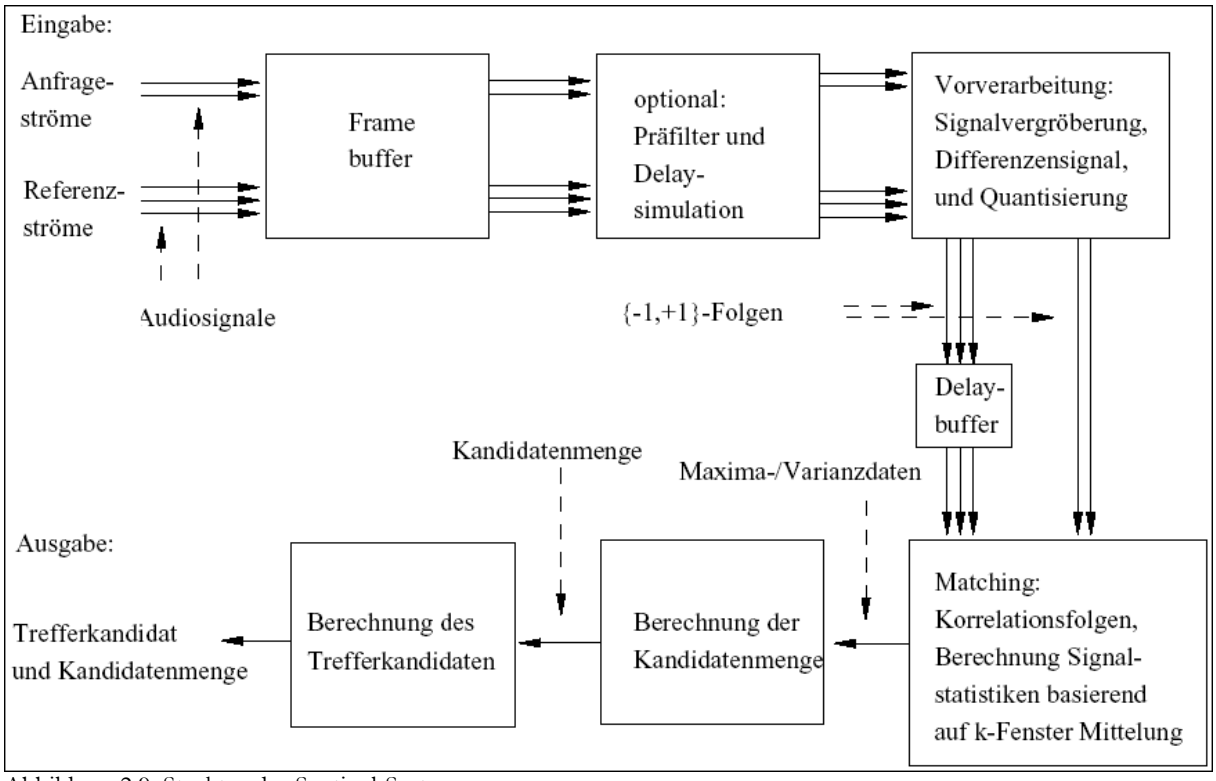


Abbildung 2.9: Struktur des Sentinel-Systems

Ein implementiertes Audiomonitoringsystem ist das von F. Kurth und R. Scherzer entwickelte Sentinel-System [58]. Aufsetzend auf dem in Abschnitt 2.5.5 beschriebenen Audentify!-System [57] erlaubt diese C++-Software den Echtzeit-Vergleich einer Menge eingehender Audioströme mit in einer Datenbank repräsentierten Referenz-Audiostücken, beispielsweise im Szenario der Suche nach im Radio gespielten Musikstücken, die über ein Mobiltelefon angefragt werden. Nach einer optionalen Vorverarbeitung mittels eines linearen Bandpass-Filters, durch den eine Simulation eines rauschenden Kanals erreicht werden kann, können zu jedem Anfrageelement verzögerte Kopien in den Eingabestrom eingefügt werden. Dadurch wird die Simulation eines Halleffektes ermöglicht. Die Merkmalsextraktion erzeugt aus den gefensterten Signalen eine Folge von Differential-Signalen und quantisiert diese zu einer {–1, +1}-Sequenz. Der frameweise Vergleich der Anfrage mit den ebenfalls aus {–1, +1}-Strömen bestehenden Referenzströmen – und zusätzlichen verzögerten Versionen davon – erfolgt über eine Korrelationsfunktion, deren über k Frames gemittelte Amplituden-Spitzenwerte in Verbindung mit zur Diskriminierung von konstanten Plateaus und echten Spitzenwerten verwendeten Varianzwerten eine Menge von möglichen Trefferkandidaten ergeben. Daraus wird dann ein sogenannter *Superkandidat* gewonnen, der zusammen mit den übrigen Kandidaten als Systemausgabe zurückgeliefert wird. Die Autoren evaluieren in ihrer Veröffentlichung neben möglichen maximalen Verzögerungseinspeisungen und Verzerrungstoleranzen auch die maximale Anzahl sowohl simultan verarbeitbarer Referenz- wie separat auch Anfrageströme bei jeweils fixen übrigen Systemparametern und weisen so die Praxistauglichkeit des Sentinel-Systems nach [58].

Das von Garcia et al. entworfene AIDA-Audiomonitoringsystem [41] verwendet eine hierarchische Struktur auf Basis von CORBA und Web Services, dessen Knoten jeweils versuchen, eine gegebene Anfrage zu identifizieren und bei Misslingen den nächsthöheren Knoten aufrufen. In der Identifikationsschicht wird die von Cano, Batlle et al. [22] entwickelte AudioDNA-Technik in Form von C++-Bibliotheken eingesetzt. Zwei in CORBA umgesetzte Schichten organisieren die Zuteilung der Identifikationsaufträge an physikalisch dezentral gehaltene Knoten, die zur Vereinheitlichung in eine Menge *virtueller Knoten* eingeteilt sind. Die

Anbindung an Clients, die verschiedene Betriebssysteme und Anforderungen haben können, wurde über SOAP/Web Services umgesetzt. Durch die dezentrale Lösung ist dieses System hervorragend skalierbar und kombiniert dazu die guten Performanz- und Robustheitseigenschaften des AudioDNA-Verfahrens. Ein Prototyp dieses Systems wird in Spanien zur Überwachung mehrerer Radio- und Fernsehsender eingesetzt.

Neben diesen datenbankbasierten Systemen ergeben sich aus den ständig fortschreitenden Technologien auch immer alltäglichere Verwendungsmöglichkeiten. So sind erste Sicherheitssysteme zur robusten Klassifikation von akustischen Signalen in *alltägliche* und *verdächtige* Geräusche auf dem Markt, die über ein angeschlossenes Mikrofon einen Einbruch beispielsweise anhand des Geräusches von splitterndem Glas erkennen und Alarm schlagen können. Ein ähnliches System ist auch als Notrufsystem im medizinischen Bereich oder für Kleinkinder denkbar.

Nooralahiyan et al. entwickelten ein kostengünstiges System zur Überwachung des Straßenverkehrs über ein verteiltes sensorisches Netzwerk. Mit Hilfe der Eingaben von gerichteten Mikrofonen und der Analyse durch eine Autokorrelationsfunktion erlaubt dieses System die Klassifikation des Verkehrsgeräusches in verschiedene Fahrzeugtypen. Ein Neuronales Netz lernte anhand von Trainingsdaten die Diskriminierung in vier verschiedene Fahrzeugklassen und erreicht eine überragende Klassifikationsleistung, die auch künstliche Verzerrungen nicht wesentliche beeinträchtigten [80]. Duarte und Hu entwickelten diese Überwachungstechnik weiter und verwenden in ihrem Ansatz ein System aus drahtlos kommunizierenden verteilten Sensoren, die über eine Reihe von Klassifikationsverfahren Umgebungsgeräusche einordnen [33]. Im Forschungsfeld des Verkehrs-Monitorings arbeiten darüber hinaus weitere Forschungsgruppen, zu deren Zielen unter anderem die Steigerung der Transporteffizienz, sowie die Überwachung von Umwelt- und Sicherheitsaspekten gehören [102].

Das im vierten Kapitel eingehend vorgestellte generische Monitoringsystem, das im Rahmen dieser Diplomarbeit entwickelt wurde, weist einige Gemeinsamkeiten mit den gerade vorgestellten Audiomonitoringsystemen auf. Im Gegensatz zu ihnen handelt es sich dabei jedoch um ein Konzept, das die Einbindung prinzipiell beliebiger Audiomonitoringverfahren ermöglicht. Da dieses Monitoringsystem intensiven Gebrauch von *Multimedia-Framework*-Mechanismen macht, soll das Konzept des Multimedia-Frameworks im nachfolgenden dritten Kapitel behandelt werden.

Kapitel 3

Multimedia-Frameworks

Mit steigender Prozessorleistung, wachsender Anzahl von Breitbandverbindungen und stetig stärkerer Einbindung von Computern in alltägliche Bereiche spielen Multimedia-Applikationen mittlerweile für viele Benutzer eine zentrale Rolle im Umgang mit Computern. Auch kommerziell ist die Einbindung von Multimedia-Fähigkeiten integraler Bestandteil vieler Systeme. Ausdruck für diese Veränderung war auch die Wahl des Wortes *Multimedia* zum Wort des Jahres 1995 durch die Gesellschaft für deutsche Sprache (GfdS) [43]. Multimedia besitzt keine bindende allgemeingültige Definition, sondern wird in verschiedenen Wissenschaftsgebieten unterschiedlich aufgefasst. Im technischen Bereich kann es aber mit „Synthese und Nutzung verschiedener Medien" umschrieben werden [48]. Ein weiterer Hinweis für die wichtige Rolle von Multimedia ergibt sich aus der Art, wie Menschen multimediale Daten filtern: So können die meisten Menschen nur 20% der gesehenen und separat 30% von gehörten Informationen speichern, im Fall des simultanen Reizens beider Sinne steigt die Rate des Gespeicherten dagegen auf 50% [101].

Die gestiegene Popularität von Multimedia-Anwendungen hat Auswirkungen auf sowohl Produzenten als auch Konsumenten. Da zum einen Entwicklungszyklen immer kürzer und Entwicklungsbudgets zunehmend gekürzt, zum anderen multimediale Anwendungen immer komplexer werden, hat sich der Einsatz einheitlicher, objektorientierter *Multimedia-Frameworks* bewährt, die naht- wie problemlos in System-Architekturen eingefügt werden können.

Ziel dieses Kapitels ist es, Aufgaben und Fähigkeiten verschiedener Multimedia-Frameworks herauszuarbeiten und gegenüberzustellen. Grund dafür ist die Tatsache, dass das im Rahmen dieser Diplomarbeit entwickelte Audiomonitoringsystem maßgeblich auf einem der vorgestellten Multimedia-Frameworks aufsetzt. Zu diesem Zweck wird in Abschnitt 3.1 zunächst das Konzept des Frameworks allgemein erläutert, bevor in Abschnitt 3.2 elementare Aufgabenbereiche besprochen werden. Den Abschluss dieses Kapitels bildet in Abschnitt 3.3 die Darstellung aktueller Multimedia-Frameworks, deren jeweilige Arbeitsweise zusammen mit dem entsprechenden Funktionsumfang erläutert wird.

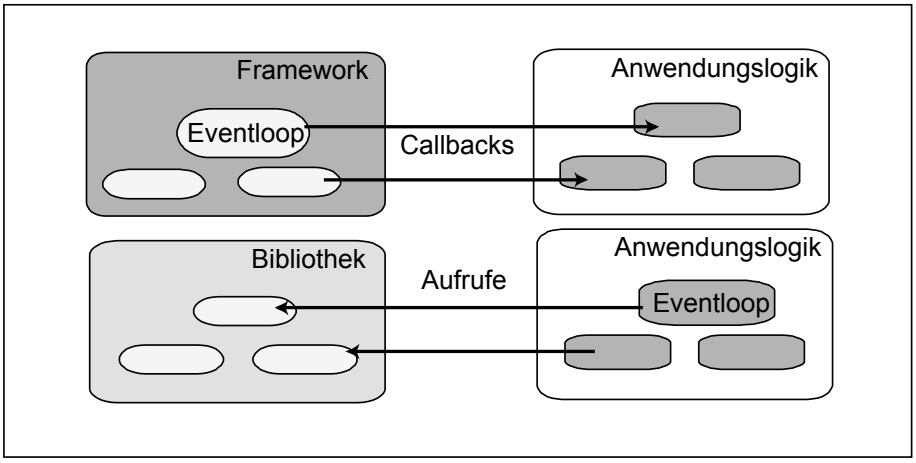


Abbildung 3.1: Strukturunterschiede zwischen Framework und Bibliothek

3.1 Frameworks

„Ein Framework ist ein Verbund von Klassen, die einen abstrakten Lösungsentwurf für eine Familie verwandter Probleme darstellen." (R.Johnson [54])

Objektorientierte Frameworks (zu deutsch etwa „Anwendungsgerüst") bieten allgemein ein implementiertes Grundgerüst aus Architektur, Code und Entwurfsprinzipien für ähnliche Applikationen eines Anwendungsbereiches, das Entwickler üblicherweise durch Vererbungsmechanismen und Instanziierung verwenden können. Im Unterschied zu konventionellen Klassenbibliotheken liegen also gewissermaßen bereits „halbfertige" generische Applikationen eines Anwendungsbereiches vor, die lediglich um eigene Applikationslogik erweitert werden müssen. Nach Fayad und Schmidt [36] weisen objektorientierte Frameworks vier primäre Vorzüge auf:

- **Modularität:** Durch die Verkapselung der Implementierung hinter öffentlichen Interfaces erhöht sich die Software-Qualität, da sich Verständnis und Wartung vereinfachen.

- **Wiederverwendbarkeit:** Die durch stabile Interfaces bereitgestellten generischen Komponenten beinhalten bereits komplette Lösungen typischer Probleme des jeweiligen Anwendungsbereiches in Form von Expertenwissen. Durch Verwenden dieser Lösungen erhöhen sich neben der Entwickler-Produktivität auch Qualität, Performanz, Verlässlichkeit und Interoperabilität.

- **Erweiterbarkeit:** Durch Erweiterung der Interfaces und durch Hook-Methoden sind Frameworks erweiterbar.

- **Invertierung der Kontrolle:** Der Applikationscode wird vom Framework aufgerufen. Dies wird auch als *Hollywood-Prinzip* bezeichnet: „Don't call me, I'll call you".

Weitere Vorteile bei Verwendung von Industriestandard-Frameworks:

- **Ergonomie:** Die Homogenität von Applikationen gleicher Frameworks erleichtert den Einstieg und Umgang mit solchen Applikationen.

- **Zuverlässigkeit:** Durch den hohen Verbreitungsgrad sind solche Frameworks vielfach und dauerhaft getestetet.

- **Performanz:** Üblicherweise weisen weitverbreitete Frameworks eine gute Leistungsfähigkeit auf.

- **Systematische Tests:** Frameworks bieten gute Möglichkeiten für den Einsatz systematischer Tests.

Während bei der Verwendung von Bibliotheken die Klassen unabhängig voneinander verwendet werden können und die Applikationsarchitektur frei wählbar ist, gehören die Klassen eines Frameworks zu einem kooperierenden Verband und beruhen auf einer vordefinierten Architektur. Daraus ergeben sich nach Fayad und Schmidt [36] einige Herausforderungen bei der Entwicklung und Verwendung von Frameworks, vor allem auch im direkten Vergleich mit anderen, weniger komplexen Ansätzen wie Patterns, Klassenbibliotheken und Komponenten:

- **Hoher initialer Entwicklungsaufwand**: Dieser ergibt sich aus der hohen Komplexität von Frameworks.

- **Lernkurve:** Die oft mehrmonatige Einarbeitungszeit in ein Framework amortisiert sich üblicherweise nur, falls Applikationen des selben Anwendungsbereiches oder Anwendungen mit übergeordneten Gemeinsamkeiten (und Unterschieden nur im Detail) entwickelt werden sollen.
  „The most profoundly elegant framework will never be reused unless the cost of understanding it and then using its abstractions is lower than the programmer's perceived cost of writing them from scratch". (G.Booch [18])

- **Integrierbarkeit:** Die Kombination verschiedener Frameworks (GUI, Datenbanken, Kommunikation, ...) ist oft sehr schwierig, vor allem, wenn auch Klassenbibliotheken oder Komponenten miteingebunden werden müssen. Die Gründe dafür finden sich auf mehreren Abstraktionsebenen, etwa wegen der oben genannten. „Invertierung der Kontrolle" im Zusammenspiel der jeweiligen Kontrollmechanismen.

- **Wartbarkeit:** Aufgrund der Komplexität von Frameworks sind Änderungen der Anforderungen oft schwer umzusetzen oder resultieren in weitreichenden Folgen für darauf aufbauende Applikationen.

- **Validierung und Entfernen von Fehlern:** Durch die zunehmende Abstraktion und das Verwenden generischer Komponenten wird eine Validierung ebenso immer schwieriger wie die Lokalisierung eines Fehlers in Framework oder Applikation. Auch die invertierte Kontrolle erschwert das Debuggen maßgeblich.

- **Effizienz:** Der Preis hoher Flexibilität ist oft eine reduzierte Effizienz, z.B. durch das übliche Verwenden von *Dynamic Binding*.

- **Fehlende Standards unter Frameworks:** Dies betrifft sowohl Design und Implementierung wie Dokumentation und Adaption von Frameworks.

Bekannte Frameworks sind beispielsweise die Microsoft Foundation Classes (MFC), ein GUI-Framework, und CORBA, ein Object-Request-Broker (ORB)-Framework, das Kommunikation

zwischen lokalen und entfernten Objekten ermöglicht. Andere ORB-Frameworks sind DCOM und das Java RMI.

Frameworks lassen sich nach dem Taligent-White-Paper [103] anhand von zwei Kriterien klassifizieren; zum einen nach Einsatzart:

- **Anwendungs-Frameworks** (engl. a*pplication frameworks*) umfassen die gesamte Anwendung, d.h. Expertenwissen zur Systemarchitektur, z.B. GUI-Frameworks.

- **Bereichsspezifische Frameworks** (engl. *domain frameworks*) beinhalten Expertenwissen zu einem speziellen Anwendungsbereich. Darunter fallen auch Multimedia-Frameworks.

- **Infrastruktur-Frameworks** (engl. *support frameworks*) stellen Systemdienste bereit, z.B. für den Dateizugriff.

Klassifikation nach Architektur:

- **architekturgetriebene Frameworks** (engl. *architecture-driven*) werden angepasst durch Vererbung und Überdefinieren. Sie beinhalten komplexe Klassenhierarchien, so dass zum einen ein hoher Einarbeitungsaufwand entsteht und zum anderen relativ viel Code zu schreiben ist. Dafür bieten sie einen hohen Grad an Flexibilität.

- **datengetriebene Frameworks** (engl. *data-driven*) dagegen werden anhand von Objektkonfigurationen und anderen Parametereinstellungen angepasst und sind daher leicht zu adaptieren. Allerdings ist das Systemverhalten dadurch stark eingeschränkt und unflexibel.

Gut entworfene Frameworks bieten üblicherweise eine zweischichtige Architektur aus einer architekturgetriebenen Basis mit einer darauf aufsetzenden datengetriebenen Schicht.

Die weitverbreitete Einteilung von R.Johnson [54] ähnelt dieser Klassifikation sehr: Nach diesem Ansatz entsprechen architekturgetriebene Frameworks in etwa sogenannten White-Box-, datengetriebene dagegen Black-Box-Frameworks. In diesem Zusammenhang wird auch von „Anpassungs-“ bzw. „aufrufenden“ (engl. *calling*) auf der einen und „Use-As-Is-“ bzw. „aufgerufenen“ (engl. *called*) Frameworks auf der anderen Seite gesprochen. Reale Frameworks sind in der Praxis normalerweise zwischen den Extremen angesiedelt und werden daher als „grau“ bezeichnet.

3.2 Multimedia-Frameworks

Multimedia-Frameworks stellen Infrastruktur und Dienste für Multimedia-Anwendungen zur Verfügung. Dies umfasst [113]:

- **Performante und ressourcenschonende Wiedergabe von Multimediadaten verschiedener Typen:**
  In jedem Multimedia-Bereich und in Abhängigkeit der betrachteten Plattform gibt es verschiedene Dateiformate, die mehr oder weniger etabliert sind. Neben der Unterscheidung zeitbasierter (kontinuierliche Medien, z.B. Video und Audio) und nicht zeitbasierter (diskrete Medien, z.B. Grafik, Text) Formate lässt sich weiterhin eine Einteilung in atomare (z.B. Bitmaps) und zusammengesetzte bzw. integrative (z.B.

HTML, PDF) Datentypen vornehmen. Ein wichtiges Gütekriterium für Multimedia-Frameworks sind die Menge und Art der unterstützten Dateiformate und die Möglichkeiten zur Konvertierung dieser Formate.
Die Effizienz des Frameworks spielt eine ebenso wichtige Rolle. Diese hängt von vielen Faktoren ab, darunter dem Grad der Abstraktion, bzw. der Anzahl von Schichten zwischen Ausführungsebene und Hardware, und der verwendeten Transportarchitektur.
Diese beruht üblicherweise auf einer hierarchischen Kombination von Modulen mehrerer Typen [28]: *Quellen* dienen der Eingabe von Datenströmen – also z.B. Module zum Einlesen von Videodateien – während *Renderer* mit der Ausgabe von Daten beschäftigt sind. Module, die auf den Multimediadaten arbeiten, werden als *Transformatoren* bezeichnet; letztere haben je mindestens einen Ein- und Ausgang. Die Modellierung solcher einzelner *Geräte* erfolgt mittels eines *Datenflussgraphen*, in dem die Geräte durch Knoten und Datenströme durch gerichtete Kanten dargestellt werden [100].
Datentransportarchitekturen lassen sich unterscheiden in Push- und Pullmodelle, je nachdem, ob der Produzent oder der Konsument den Transport initiiert. Die genauen Details der Umsetzung der Architektur bestimmen somit maßgeblich die Performanz des Frameworks.

- **Netzwerkfunktionalität & Streaming:**
  Streaming erlaubt – auch lokal – die Echtzeit-Übertragung von Multimediadaten ohne vorheriges Laden der gesamten Datei. Dies spielt vor allem in zeitkritischen Applikationen wie Live-Übertragungen und Internet-Radio eine entscheidende Rolle, beinhaltet jedoch einen durch begrenzte und/oder schwankende Bandbreite begründeten Trade-Off zwischen Qualität, Robustheit und Datenmenge. Die Daten dieser zentralen Framework-Komponente werden dabei paketweise übertragen, um – in Verbindung mit geeigneten Protokollen – eine möglichst geringe Latenz zu gewährleisten. Darüber hinaus spielt der Umgang mit verlorenen Daten eine wichtige Rolle und sollte durch entsprechende QoS-Maßnahmen behandelt werden.
  Neben einer Unterstützung grundlegender Netzwerk-Protokolle wie HTTP ist vor allem die Unterstützung von Protokollen wie RTP (Realtime Transport Protocol) [88] oder RTSP (Realtime Streaming Protocol) [89] von Interesse, die speziell für den Echtzeit-Einsatz in Multimedia-Szenarien entworfen wurden. Insbesondere die Verwendung möglichst effizienter und performanter Protokolle ist von Belang; so ist in realen Applikationen meist statt des Paket-quittierenden TCP das schnellere UDP-Protokoll implementiert, das keinerlei Garantien für das Ankommen versendeter Pakete gibt. Die daraus resultierende fehlende Fehlerkontrolle wird z.B. durch das aufsetzende RTP-Protokoll in Form von Zeitstempeln und Sequenznummern gewährleistet, das durch RTCP (RTP Control Protocol) [88] um QoS-Funktionen wie Statusinformationen der Clients, Metadaten-Transport und eindeutige Teilnehmeridentifikation ergänzt wird.
  Neben punktweisen Unicast-Verbindungen existieren mit Broad- und Multicast-Konzepten Ansätze, die komplexere Verbindungsstrukturen erlauben. In diesem Zusammenhang spielen vor allem Last-balancierende Konzepte wie IP-Multicast ein Rolle, die verfügbare Bandbreite weniger stark belasten. Auch der Einsatz von dynamischen Cachingtechnologien erlaubt eine Reduktion der Netzbeeinträchtigung [100].

- **Synchronisation von Datenströmen:**
  Dies beschreibt das Herstellen eines Gleichlaufes zwischen mehreren Prozessen zu integrativen Zwecken; so dürfen beispielsweise zusammengehörige Audio- und Videostream-Timecodes einer Fernsehübertragung nur um einen gewissen Betrag variieren, um nicht als fehlerhafte Darstellung wahrgenommen zu werden. Üblicherweise betrachtet man „weiche“ Synchronisationsanforderungen, die eine abstufbare Güte der

Synchronizität verwenden. Dem zugrunde liegt die Erkenntnis, dass das Wahrnehmen von Asynchronizität problemspezifisch ist. Daraus resultierende Informationen beeinflussen daher die Anforderungen zu zeitlichen Auflösungen der Multimedia-Frameworks. Steinmetz [100] zeigt dazu ausführliche Fallstudien aus verschiedenen Multimediabereichen und –Anwendungen, entsprechende maximale Versatzgrößen und Lösungsansätze, etwa durch QoS-Mechanismen, die der Anpassung von Puffergrößen und Latenzen dienen.

- **Bereitstellungen von Quality-of-Service(QoS)-Funktionen:**
  Zur Garantie einer gewissen Dienstgüte werden Ressourcen durch dynamische Skalierung und Reservierung entsprechend gesteuert. Vorgaben betreffen z.B. Echtzeit-Anforderungen, Korrektheit bzw. Fehlertoleranzen und garantierte Antwortzeiten. Üblicherweise sind diese nach dem ISO-Standard QoS oder weitergehenden Verfahren flexibel parametrisierbar und schichtweise implementiert [100]. Darüber hinaus werden Möglichkeiten zur Erkennung von und dem robusten adaptiven Umgang mit Fehlern gegeben, um wahrnehmbare Effekte zu vermeiden. Zusätzlich werden oft auch statistische Daten verfügbar gemacht.

- **Integration mit und Unabhängigkeit von Hardware:**
  Dies wird möglich z.B. durch ein geeignetes Treibermodell und die Unterstützung von CPU-Multimedia-Erweiterungen.

- **Verwaltung von Framework-Komponenten:**
  Üblicherweise erweitern Drittanbieter ein gegebenes Framework über eine standardisierte *Plugin*-Schnittstelle. Plugins sind im Allgemeinen kleine, portable Code-Objekte, die über einen definierten Registrierungs-Mechanismus in ein Framework integriert werden können und – ähnlich einer Bibliothek – ein oder mehrere Interfaces implementieren. Das Einbinden bzw. das Aufrufen des enthaltenen Codes findet dann über Funktionen des Frameworks statt, z.B. unter Verwendung eines Factory-Patterns. Beispiele für Plugins sind Audio-/Video-Codecs und Filter-Transformatoren.

- **Dienste für die Produktion von Multimediadaten:**
  Bereitstellung von entsprechenden Werkzeugen und dokumentierten Schnittstellen, z.B. für das *Capturing* von Daten, also die Aufnahme eines Eingangsdatenstromes, mit entsprechender Dokumentation und automatischen Metadaten.

- **Rechtemanagement und Sicherheitsfunktionalität:**
  Hierunter fällt neben einer Sicherheit gegenüber Angriffen durch kryptografische und konfiguratorische Maßnahmen auch Ausfallsicherheit in Form von Datensicherungsmechanismen, Threadsicherheit und Robustheit gegenüber z.B. fehlerhafter Hardware. Die rechtlichen Aspekte dagegen betreffen die Einhaltung von Copyright-Verfahren und damit verbundene Zugriffsbeschränkungen bzw. -rechte, Datenschutz & Anonymität, Funktionen zum Authenzitätsnachweis, Kryptoverfahren zur Vertraulichkeitseinhaltung und Integritätsprüfungen für Multimediadaten [100].

- **Plattformunabhängigkeit:**
  Bereitstellung standardisierter Komponenten und Schnittstellen. Ein höheres Maß an Plattformunabhängigkeit geht allerdings üblicherweise einher mit einer stärkeren Abstraktion und damit oft mit einer geringeren Performanz.

- **Medienserver-Funktionen:**
  Speicherung und Verteilung multimedialer Daten, verbunden mit einem Verzeichnisdienst. Je nachdem, ob Client oder Server den Sendeprozess initiieren, spricht man von Pull- oder Push-Server. Pull-Server werden im Allgemeinen nur in LANs eingesetzt, Push-Server dagegen auch für Broad- oder Multicast-Dienste, z.B. im Internet.

- **Bereitstellung von Interfaces zur flexiblen Erweiterung:**
  Neben High-Level-Ansätzen wie Skriptsprachen ist vor allem der systemweite objektorientierte Eingriff von der Möglichkeit z.B. neue Dateitypen hinzuzufügen bis hin zu Low-Level-Interfaces interessant. Neben der generellen Dokumentation verwendeter Interfaces ist dazu vor allem die Bereitstellung von Entwicklungspaketen (engl. *Software Development Kit*, *SDK*) von primärem Belang.

3.3 Aktuelle Multimedia-Frameworks

Die große Marktbedeutung von Multimedia-Frameworks führt dazu, dass in den meisten kommerziellen Fällen die Software zum Abspielen (*Player*) kostenlos verfügbar ist, um eine starke Marktdurchdringung zu erlangen, die zur vollwertigen Produktion benötigten Versionen und Servereditionen jedoch dafür um so kostspieliger sind.

Zwar existieren auch zahlreiche Multimedia-Frameworks, die wie z.B. MME [29], das an der Universität Zürich entwickelte MET++ [1], das Berkeley CMT [97] und Nsync [12] aus universitärem Kontext heraus oder wie Steinbergs VST-Technologie [123] oder Cycling 74 Max/Msp [26] zumindest teilweise kommerziell entwickelt wurden, im Folgenden soll jedoch lediglich eine Auswahl weit verbreiteter oder zumindest hinreichend komplexer Multimedia-Frameworks genauer vorgestellt werden.

3.3.1 Quicktime

Apple schuf mit Quicktime [7] 1991 als Erweiterung des eigenen Betriebssystems zur Behandlung zeitbasierter Daten die erste Streaming-Technologie, die heute in Form der stark weiterentwickelten Version 6 elementarer Bestandteil der aktuellen „OS X"-Systemarchitektur ist. Auf Quicktime basierende Software kann darüber hinaus auch für Windows und Java entwickelt werden. Zum umfassenden Funktionsumfang gehören unter anderem die Anzeige und Bearbeitung von Audio, Video, Grafik, Text, MIDI, Flash und VR-Panoramen sowie mehr als 250 unterstützter Dateiformate. Das enthaltene Streaming basiert auf der Unterstützung von HTTP, RTP, RTSP und 20 weiteren Netzwerkprotokollen und bekommt neben Broadcasting auch durch die ebenfalls enthaltene Funktionalität des Mobilfunk-Standards 3GPP zusätzliche Gewichtung.

Die Quicktime-Architektur besteht aus einer Kombination flexibler Toolsets und Plugin-Komponenten in Form von über 2000 Funktionen. Zu diesen Toolsets gehören z.B. die grundlegende Movie Toolbox, die Streaming API und andere Toolsets zu jeweils abgegrenzten Funktionsbereichen wie Bildkompression und Videocapturing, die von Applikationen explizit aufgerufen oder bei Bedarf automatisch verwendet werden und über Callbacks mit diesen kommunizieren. Gleiches gilt für die Plugin-Komponenten, zu denen beispielsweise Handler für die einzelnen Medientypen, Standard-User-Interfaces, und Daten-Handler für verschiedene Arten von Datenquellen wie lokale Dateien, URLs oder Zeiger gehören. Jede Komponente wird dabei über einen je vierstellig repräsentierten Typ, Subtyp und Herstellercode identifiziert und über den *Component Manager* angesprochen.

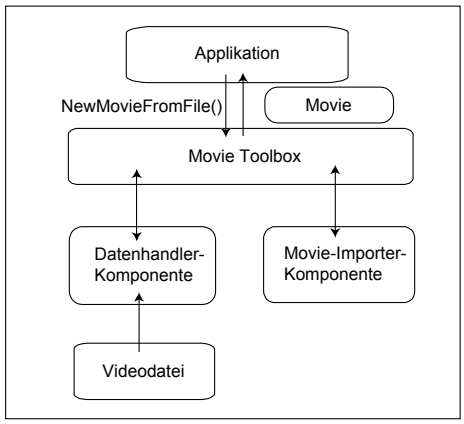


Abbildung 3.2: Beispiel zur Quicktime-Architektur

Im Beispiel aus Abbildung 3.2 fordert die Applikation die Movie Toolbox auf, eine bestimmte Datei zu öffnen. Die Toolbox verwendet daraufhin eine zum Dateiformat entsprechende Dateihandler-Komponente, um die Datei anzusprechen. Falls es sich bei der Datei nicht um eine (native) Quicktime-Videodatei handelt, wird unter Verwendung der zum Dateiformat passenden Movie-Importer-Komponente ein Movie-Objekt aus der Datei erzeugt. Dieses Objekt wird an die Applikation zurückgeliefert.

3.3.2 Java Media Framework

Das Java Media Framework (JMF) [53] stellt als optional installierbarer Teil der Java Media API umfangreiche Funktionalität im Umgang mit zeitbasierten Daten bereit. Dies umfasst in der aktuellen Version 2 unter anderem Capturing, Datenverarbeitung, Streaming via RTP/RTCP und ein flexibles Plugin-Modell. Großer Vorteil der JMF-Architektur ist die zugrunde liegende Plattformunabhängigkeit mit gleichzeitiger Unterstützung der gängigen Dateiformate. Aus Performanzgründen steht sowohl eine plattformübergreifende Pure-Java-Variante als auch „Performance Packs“ mit nativem Code für verschiedene Plattformen bereit.

JMF verwendet ein fundamentales Datenverarbeitungsmodell aus Eingabe-, Verarbeitungs- und Ausgabeschicht mit den Schlüsselelementen *Capturing Device/Data Source*, *Player/Processor* und *Data Sink*, das über die Implementation von Interfaces mit Hilfe von sogenannten „Managern“ erweitert werden kann. JMF kennt vier verschiedene Typen von Managern, die ihre Daten mittels einer Registrierungsdatenbank verwalten:

- **Manager:** Handhabt die Erzeugung von JMF-Klassen
- **PackageManager:** Verwaltet verfügbare JMF-Klassen
- **CaptureDeviceManager:** Verwaltet die verfügbaren Capturing-Geräte
- **PluginManager:** Verwaltet verfügbare Plugins

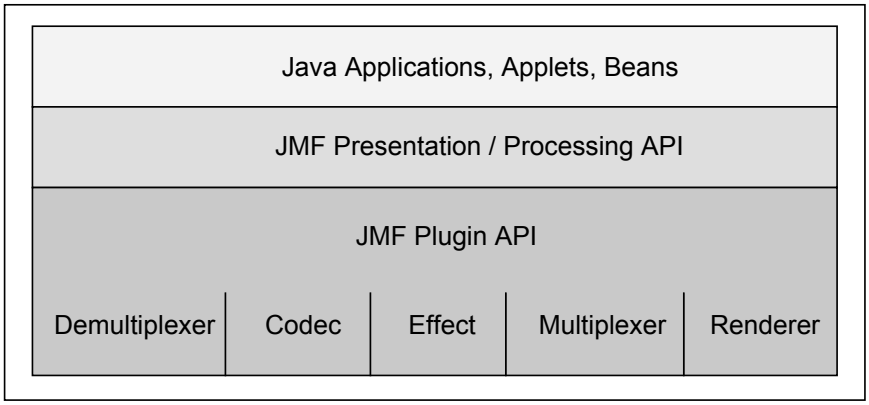


Abbildung 3.3: High-Level-Architektur von JMF

Die Plugin-Architektur umfasst fünf verschiedene von der Player- oder der Processor-Klasse abgeleitete Typen:

- **Demultiplexer:** Parsing und Zerlegung von Eingabeströmen, z.B. WAV-Daten;
- **Codec:** (De-)Komprimierung und Konvertierung von Daten;
- **Effect :** Spezialisierter Typ von Codec, der sonstige Transformationen durchführt;
- **Multiplexer:** Vereinigung von Datenströmen, z.B. in ein MPEG-Format;
- **Renderer:** Auslieferung von Daten;

Plugins werden beim PluginManager unter Verwendung eines eindeutigen Bezeichners registriert, der üblicherweise einem umgekehrten Domain-Namen des Autors folgt, z.B. „COM.sun.java.MeinEffekt“. Hinzu kommen Angaben zu unterstützten Datenformaten.

Sämtliche Objekte, z.B. Zeitstempel, von *Controller*-Objekten verwaltete Events, Datenblöcke und Objekteigenschaften werden ebenso über eine Hierarchie von Interfaces beschrieben und ausgelesen wie Medienrepräsentationen und -handler. Erweiterungen von JMF finden daher entweder per Plugin oder per direkter Implementation der Player-, Processor-, DataSource- oder DataSink-Interfaces statt.

3.3.3 Linux

Der Stand der Entwicklung von Multimedia-Frameworks zeigt ein uneinheitliches Bild: Linux als Multiusersystem und nicht als Hochleistungssystem für Einzelrechner hat historisch einigen Rückstand zu den professionellen Multimedia-Frameworks anderer Betriebssystem und besitzt (noch) keinen Distributions-übergreifenden Standard. Deutlich wird dies anhand der Bestplatzierten in der Rubrik „Best Multimedia Framework“ des Linux New Media Award 2004 [60]: Die Wertungen der drei Gewinner *JACK*, *GStreamer* und *SDL* unterscheiden sich um nicht einmal 5%.

JACK [51] ist ein auf dem populären ALSA-Audio-Treiber-Format [4] aufsetzender Audio-Server mit geringer Latenz, der Streaming, Callbacks und Inter-Applikations-Kommunikation unterstützt. Das aus dem Umfeld des Gnome-Projekts heraus entstandene *GStreamer* [44] ist eine in Anlehnung an Microsofts nachfolgend erläutertes DirectShow-Framework entwickelte Streaming-Media-Architektur, die für In-Prozess-Konstruktion von Media-Pipelines konzipiert wurde und ein eigenes Modell zur Verwaltung von Plugins, hier *GstElements* genannt, beinhaltet.

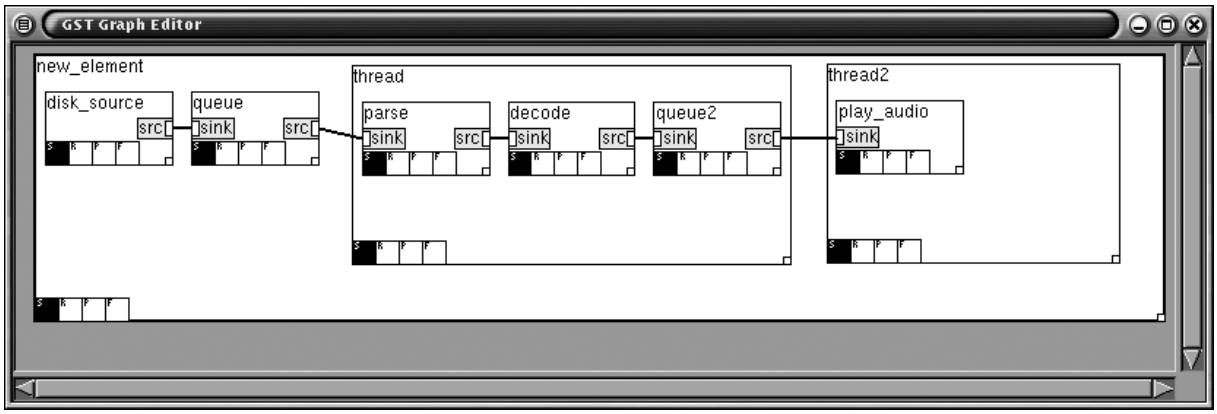


Abbildung 3.4: GStreamer-Pipeline zum Abspielen einer MP3-Datei

Diese werden wie in Abbildung 3.4 dargestellt in Pipelines, also einer seriellen Verschaltung von Modulen, organisiert, indem die für den Austausch von Datenblöcken zuständigen Ein- bzw. Ausgabeinterfaces, *Pads* genannt, miteinander verbunden werden. GStreamer wurde als Open-Source-Projekt auch auf andere Plattformen portiert. Die *Linux Simple Direct Layer* (SDL) [61] schließlich ist ebenfalls plattformübergreifend und setzt unter Windows beispielsweise auf DirectX auf. Es unterstützt neben Audio auch Video, Ereignisbehandlung und Threading-Funktionalität wie Semaphore und Mutex-Objekte. Es scheint etwas fragwürdig, in allen drei Fällen sowohl von *Multimedia* als auch *Frameworks* zu sprechen: JACK unterstützt lediglich Audio-Applikationen und ist wie auch SDL wohl eher als Bibliothek denn als ein echtes Framework zu bezeichnen.

3.3.4 Helix / RealMedia

Um ein vollwertiges Multimedia-Framework für Linux verfügbar zu machen, haben mit RedHat und Novell zwei führende Linux-Distributoren Pläne angekündigt [46], das aus dem kommerziellen Real-Player von Real Networks [86] heraus entwickelte plattformübergreifende Open-Source-Projekt Helix Player [46] zum Standard-Framework ihrer Distributionen zu machen. Helix ist ein auf über 1000 Interfaces und Methoden aufbauendes Multimedia-Framework, bestehend aus den Elementen *Helix DNA Server*, *Client* und *Producer*, die Backend-, Player- und Authoring-Komponenten darstellen. Die Helix-Plattform unterstützt COM-basierte Plugins, Monitoring, QoS-Dienste, webbasierte Administration, Zugriffskontrolle, Caching und vor allem Streaming-Funktionalität, deren Güte durch z.B. das „loss protection"-System zur Rekonstruktion verlorener Frames sowie weitergehende Video-Filter und Audio-Resampling-Routinen weiterentwickelt wurde. Neben HTTP und FTP werden RTSP (Realtime Streaming Protocol), das RTP Control Protocol (RTCP) für QoS-Zwecke und SDP (Session Description Protocol) zum Initiieren von Multimedia-Sessions verwendet, die ihrerseits auf TCP/IP bzw. UDP aufsetzen. Unterstützte Dateitypen sind neben RealAudio/-Video auch Apple QuickTime, MPEG-2, MPEG-4 und selbst Microsofts Windows Media Format.

3.3.5 Windows Media

Windows Media ist eine Plattform für Erstellung, Transport und Distribution von digitalen Multimedia-Inhalten, die auf den in Abb. 3.5 dargestellten Komponenten beruht und in das Windows-Betriebssystem integriert ist. Aus letzterem ergibt sich zum einen ein großer Verbreitungsgrad und zum anderen eine für andere Frameworks kaum erreichbare Performanz. Gleiches gilt auch für die speziell zugeschnittenen Codecs, die – auch in Verbindung mit weiteren

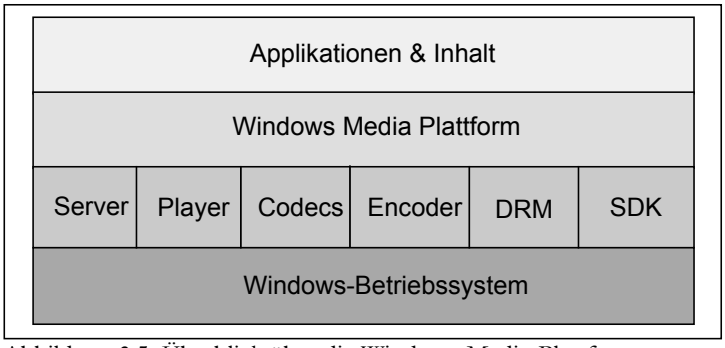


Abbildung 3.5: Überblick über die Windows-Media-Plattform

Merkmalen wie adaptiver Bitrate, *Frame Smoothing* und variabler Wiedergabe-Rate – sehr effizientes und komfortables Streaming durch die skalierbare Server-Komponente mittels HTTP, RTSP und dem Media Transfer Protocol (MTP) [66] erlauben. Die zum Streaming benötigten Elemente können ebenso kostenlos nachgerüstet werden wie der Encoder. Dieser generiert optional eine auf optimiertes Streaming ausgelegte ASF-Datei (Advanced Streaming Format), die den enkodierten Datenstrom in mehreren Qualitätsstufen enthält und auch HDTV-Ansprüchen moderner externer Multimedia-Geräte (mit entsprechend angepasstem Windows-Betriebssystem) genügt.

Der Player bietet eine durch verschiedene Skins anpassbare Oberfläche und gibt sowohl gestreamte als auch andere Multimediadaten wieder, sogar in Surround-Audio bis 7.1 Kanälen. Fehlende Codecs werden – wenn möglich – automatisch nachgeladen.

Die System-Architektur besitzt darüber hinaus ein Plugin-Modell für Player, Server und Encoder, das über ein SDK für eigene – im Vergleich zu anderen Frameworks allerdings eingeschränkte – Anforderungen, z.B. zur Synchronisation mit portablen Abspielgeräten oder für das digitale Rechtemanagement verwendet werden kann. Letzteres liegt allen Komponenten zugrunde, kann serverseitig in Echtzeit angepasst werden und erlaubt über ein eigenes Lizenzmodell fortschrittliche DRM-Operationen wie quantitativ beschränkte Abspielwiederholungen (engl. *counted play*), zeitliche Gültigkeitsbereiche und Restriktion bezüglich der weiteren Verwendung von Inhalten, z.B. bezüglich des Brennens auf CD oder den Transport auf portable Medien. Darüber hinaus bieten im Player integrierte QoS-Funktionen z.B. Feedback über Bandbreite, Mediennutzung und Paketverluste.

3.3.6 DirectX

Microsoft DirectX [67] ist eine 1995 eingeführte performante Multimedia-Erweiterung des Windows-Betriebssystems, bestehend aus Low-Level-APIs für z.B. 2D- und 3D-Grafik, Soundeffekte und Musik, Eingabegeräte und vernetzte Applikationen. DirectX existiert aktuell in der Version 9 und bietet abstrakten Zugriff auf Hardware-Merkmale.

Mit DirectShow ist eine Komponente zur Erstellung von Streaming-Media-Anwendungen enthalten, die umfassende Multimedia-Funktionalität und flexible Erweiterbarkeit bietet: Unterstützt werden grundsätzlich ebenso Wiedergabe und Capturing von Datenströmen gängiger Formate wie Konvertierungen und andere Transformationen. Der Funktionsumfang, z.B. vorhandene Netzwerk-Protokolle, hängt von den im System registrierten Plugins ab. Standardmäßig werden jedoch alle gängigen Anforderungen erfüllt.

Da das im vierten Kapitel vorgestellte generische Monitoringsystem, das im Rahmen dieser Diplomarbeit entwickelt wurde, maßgeblich auf DirectShow aufbaut, sollen nachfolgend einige Teile des DirectX-Frameworks besonders hervorgehoben werden.

DirectShow basiert maßgeblich auf dem Component Object Model (COM), das fester Bestandteil von Windows ist. Es besteht aus einer Menge von hierarchischen Interfaces sowie aus

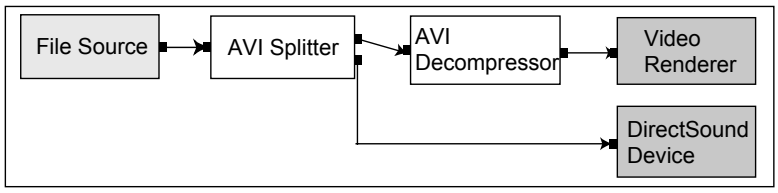


Abbildung 3.6: Filtergraph-Struktur zum Abspielen einer AVI-Datei.
Quell-, Transformatoren- und Rendererfilter sind unterschiedlich gefärbt.

einigen zentralen COM-Komponenten und einer erweiterbaren Zahl von Plugins, hier *Filter* genannt. Diese werden mit Hilfe der wichtigsten Komponente, des *Filtergraphen*, entweder automatisch oder gezielt miteinander verbunden und bilden so einen Datenflussgraphen. Der Filtergraph kontrolliert darüber hinaus sämtliche den Graphen betreffenden Belange, z.B. Synchronisationsfunktionen oder Anfragen der Applikationen bezüglich einer Kontroll-Komponente für bestimmte Wiedergabe-Merkmale.

Filter sind ebenfalls COM-Komponenten, die das CBaseFilter-Interface implementieren und anhand ihrer definierten *GUID* (*Global Unique Identifier*, ein unter COM verwendeter kurzer Textstring zur Identifikation z.B. eines Filters, Interfaces oder Datentyps) im System identifiziert, registriert und verwaltet werden. Zur Verbindung mit anderen Filtern besitzt jedes Filter besondere Teilobjekte, genannt *Pins*, die sich in Ein- und Ausgangspins unterteilen lassen und mittels in Interfaces veröffentlichten Methoden zu Format- und Typeigenschaften ihre Verbindbarkeit mit Pins anderer Filter abstimmen. Pins sind auch für den Datentransport verantwortlich, indem sie gestreamte Datenblöcke von einem Ausgangspin an einen Eingangspin weiterreichen und somit von einem Filter zum nächsten übertragen. Es gibt drei Grundtypen von Filtern, die eine entsprechend ihrem Typ verschiedene Anzahl von Ein- und Ausgangspins besitzen: *Quellfilter*, *Transformatoren* und *Renderer*. Quellfilter dienen der Eingabe von Datenströmen und haben daher nur einen Ausgangspin. Dazu gehören z.B. Capturing-Filter und Filter zum Lesen in Dateien. Transformatoren verändern und verteilen Daten und verfügen daher über mindestens je einen Ein- und Ausgangspin. Renderer schließlich besitzen nur einen Eingangspin und handhaben die Ausgabe des Datenstroms, z.B. durch Schreiben in eine Datei oder die Weitergabe an Audio- oder Videotreiberinstanzen. In Abbildung 3.6 ist ein Beispiel-Filtergraph mit den enthaltenen Filtern dargestellt.

Durch ein von Microsoft bereit gestelltes SDK ist es sowohl möglich, DirectShow um neue Filter zu erweitern als auch bestehende Filter in eigenen Applikationen zu verwenden. Diese flexible Erweiterbarkeit kann dazu genutzt werden, Quellfilter für neue Dateiformate zu entwickeln, andere Abspielmechanismen zu implementieren oder den Datenfluss vollständig zu ändern. Auch die Erstellung grundsätzlich neuer Filtertypen ist möglich. Grundlage der Filter-Entwicklung sind bestimmte Identifikationsangaben, die Implementierung von DirectShow-Interfaces und die Kompilierung als Windows-DLL. Zur Verwendung muss der im Allgemeinen kleine Filter auf dem Ziel-Betriebssystem registriert werden.

Aufgaben wie Abspielkontrolle und der *Seeking* genannte Vorgang zur wahlfreien Neupositionierung des in Quellfiltern enthaltenen Zeigers auf die aktuelle Position können über weitere zentrale COM-Komponenten durchgeführt werden. Manche dieser Operationen setzen allerdings das Vorhandensein einer Implementation bestimmter Interfaces jedes im Graphen enthaltenen Filters voraus.

DirectX bietet darüber hinaus weitere, meist in COM-Objekte/-Schnittstellen verpackte Funktionalität wie erweiterte Synchronisations- und Debuggingmechanismen, die auch in Verbindung mit DirectShow verwendet werden können. Ein Nachteil von DirectX ist das Fehlen weitergehender QoS-Funktionen.

Zwar ist DirectX bisher nur für Windows verfügbar, das mittlerweile in Cedega umbenannte WineX-Projekt [24] stellt eine Portierung von DirectX für Linux zur Verfügung, die z.B. von Spieleentwicklern zahlreich genutzt wird, um populäre Titel auch unter Linux verfügbar zu machen.

Abschließend soll auf ein weiteres Multimedia-Framework eingegangen werden, dessen Umsetzung allerdings erst noch bevorsteht, aufgrund der enthaltenen Konzepte jedoch in die hier vorgelegte Aufzählung gehört:

3.3.7 MPEG-21

Ziel des MPEG-21-Standards [71] ist die Schaffung einer vereinheitlichten Infrastruktur für Multimedia-Kommunikation und –Dienstleistungen, die allen Produzenten, Distributoren und Benutzern gleichberechtigt Homogenität, Transparenz und Kompatibilität bieten soll.
MPEG-21 beruht auf zwei Konzepten, nämlich dem fundamentalen *Digital Item*, einem strukturierten digitalen Objekt mit einer Standardrepräsentation und Metadaten, sowie der darauf aufbauenden Menge von Benutzerinteraktionen, die Erzeugung, Austausch, Transport, Verarbeitung, Konsumierung, Handel und sonstige Manipulationen des Objektes umfasst. Um dies zu ermöglichen, muss das Digital Item genau identifiziert, beschrieben, verwaltet und geschützt werden. Dies setzt eine Standardisierung sämtlicher verwendeter Komponenten, auf Interfaces fußende Inhalts-Definitionen von Syntax und Semantik sowie die Beschreibungen enthaltener Beziehungen voraus.
Das MPEG-21-Framework basiert daher grundsätzlich auf sieben Schlüsselelementen:

- **Digital Item Declaration:** Einheitliche Abstraktion zur Definition von Digital Items
- **Digital Item Identification and Description:** Mechanismen zur Identifikation und Beschreibung
- **Content Handling and Usage:** Interfaces und Protokolle für sämtliche Benutzerinteraktionen
- **Intellectual Property Management and Protection:** Rechteverwaltung
- **Terminals and Networks:** Transparenz gegenüber dem Transportmedium
- **Content Representation:** Darstellung der Inhalte
- **Event Reporting:** Performanzmessung

MPEG-21-Benutzer werden nicht in Kategorien wie etwa *Produzent* eingeteilt, sondern individuell anhand ihrer Beziehungen zu anderen Benutzern identifiziert.

Die sich durch eine solche Standardisierung eröffnenden Möglichkeiten sind offensichtlich so weitreichend, dass der gesamte Multimedia-Markt maßgeblich davon betroffen sein dürfte, vom Privatanwender über Internethandel bis zu digitalem Fernsehen.

Nachdem in diesem und dem vorigen Kapitel die Grundlagen dafür gelegt wurden, soll im nun folgenden zentralen vierten Kapitel das entwickelte generische Monitoringsystem für akustische Datenströme konzeptionell vorgestellt werden.

Kapitel 4

Entwurf des GenMAD-Systems

Vorgegebenes Ziel der vorliegenden Diplomarbeit ist die Konzeption, Realisierung und erste Evaluation eines generischen Monitoringsystems für akustische Datenströme. Das System soll in der Lage sein, unter Verwendung einer oder mehrerer Eingangsaudiodatenströme eine frei konfigurierbare Signalfluss-Verschaltung zu ermöglichen, wobei Verarbeitungsergebnisse geeignet visualisiert und als formatierte Datei ausgegeben werden können sollen. Dies beinhaltet die folgenden Aspekte:

- Verarbeitung von On- und Offline-Eingangsdatenströmen, d.h. sowohl in Echtzeit eingehende als auch in Dateien gespeicherte Audiodaten.

- Erstellung eines geeigneten Plugin-Konzeptes mit möglichst universeller, für Nachfolgearbeiten gut zugänglicher Schnittstelle, um Signalverarbeitungsmodule in das System einbinden zu können. Unterstützt werden sollen zwei Signaltypen: Zum einen „herkömmliche" zeitbasierte Daten $x : \mathbb{Z} \to X$ äquidistanter Abtastung mit impliziter Zeitachse, zum anderen durch Ereignismengen $x \subseteq \mathbb{Z} \times X$ modellierte nicht-äquidistant abgetastete Signale, deren Tupel die Angabe eines expliziten Zeitpunktes $t \in \mathbb{Z}$ enthalten. Letztere sind z.B. als Ausgabe von Identifikations- oder Klassifikationsmodulen zu verstehen und werden nachfolgend als *Events* bezeichnet.

- Die Plugins müssen durch den Benutzer geeignet konfigurierbar sein und in einer beliebigen Kaskadierung verwendet werden können. Darüber hinaus muss ein Parameterabgleich verbundener Plugins jenseits reiner Signaldaten möglich sein.

- Das eingesetzte Streamingmodell muss in der Lage sein, jedem Plugin die zu bearbeitenden Daten paketweise und mit einer geeigneten Paketlänge anzubieten.

- Mitschneiden sowohl der Audiosignaldaten als auch der Eventdaten eines Plugins in einer separaten, geeignet formatierten Datei sollen möglich sein.

- Visualisierungskomponenten für beide Signaltypen sind erforderlich.

- Exemplarische Plugin-Umsetzung eines von der betreuenden Arbeitsgruppe Multimedia-Signalverarbeitung bereitgestellten Audioidentifikations-Algorithmus und eines eigenen Klassifikators, zur Unterscheidung eines Signalausschnitts in die Klassen *Sprache*, *Musik*, *Stille* und *Sonstiges*.

- Zu implementieren sind Methoden zur geeigneten linearen Kombination von Datenströmen mehrerer Plugins, etwa zur Behandlung widersprüchlicher Identifikationen.

- Durchzuführen ist ein Test des Systems anhand der Analyse von Radiosendungen.

- Das System soll unter Windows 2000 lauffähig und in MS Visual Studio 6 kompilierbar sein.

4.1 Verwandte Ansätze & Motivation

Es existieren, wie in Abschnitt 2.7 erläutert, zahlreiche Audiomonitoring-Systeme verschiedenster Art. Diese haben jedoch gemein, dass sie jeweils auf einen speziellen Zweck hin entwickelt wurden. Dabei kann es sich um so verschiedene Ansätze wie das Verkehrsüberwachungssystem von Nooralahiyan et al. [80], das von Kurth und Scherzer entwickelte Sentinel-System [58] oder ein kommerziell zur Überwachung von Radiostationen eingesetztes System handeln. Jedes dieser Systeme setzt auf einer bestimmten Monitoringtechnik auf und verwendet eine eigens für diesen Fall entworfene Anwendungsschicht, etwa zur Bereitstellung von Visualisierungs- und Interaktionsfunktionalität. Diese Spezialisierung ist um so erstaunlicher, als die meisten Audioidentifikations- und Klassifikationsverfahren aufgrund ihrer verschiedenartigen Herkunft bestimmte Anwendungsszenarien und Umgebungsparameter besitzen, in denen sie sich durch besondere Güte auszeichnen, sei es bezüglich Robustheit, Performanz oder Effektivität. In anderen Szenarien dagegen können viele weniger überzeugen, da sie nicht universell einsetzbar sind.

Die Verschiedenheit der Systeme erschwert darüber hinaus die Einarbeitung und schränkt dadurch einen effektiven Wissensaustausch ein, was die Entwicklung von Lösungen für den Markt verlangsamt. Sicherlich spielen dabei in einigen Fällen kommerzielle Strategien eine Rolle. Hinzu kommt das in anderen Ansätzen auffällige Fehlen von eingebundenen Standards wie populären Plugin-Schnittstellen. Ein mit diesen Schnittstellen ausgestattetes System profitiert zum einen von einer solchen Erweiterbarkeit, zum anderen kann es aus einer, je nach Standard verschieden großen, Menge existierender Algorithmen schöpfen, um eine qualitative Verbesserung der enthaltenen Routinen, z.B. im Bereich der Vorverarbeitung, zu erreichen und dem Anwender mehr Handlungsspielraum zum experimentellen Verwenden neuartiger Ansätze zu geben.

Eine Möglichkeit des generischen Audiomonitorings unter Verwendung solcher Plugins besteht in der Zweckentfremdung bestehender Applikationen aus dem professionellen Audio-Bereich. Dazu bieten sich vor allem sogenannte *Digital Audio Workstations* wie Steinbergs Cubase [123] oder das von Apple entwickelte Logic [6] an, die neben einer Unterstützung zeitbasierter Audiosignale auch nicht-äquidistante, Event-basierte Daten in Form des MIDI-Formates [68] verarbeiten können. Sie verfügen über eine große Flexibilität bei der Verschaltung von signalverarbeitenden Modulen, einen großen Funktionsumfang für das Arbeiten mit Audiodaten und Ansätze zur Visualisierung der beiden genannten Signaltypen. Noch tiefere Veränderungen ermöglicht das von Native Instruments [78] entwickelte *Reaktor*, welches ein vollständig modulares System zur Verarbeitung und Synthese von Audiodaten darstellt. Die Anforderungen für die Arbeit mit eventbasierten Identifikationsdaten überfordern jedoch die Flexibilität derzeit bestehender Systeme.

Das von Tzanetakis et al. entwickelte MARSYAS-System [64, 106] ist kein Audiomonitoring-Framework, sondern ein Framework für Entwicklung und Test von Algorithmen zur Audioanalyse und -Synthese, bietet jedoch durch die verwendete Client/Server-Struktur und den

freien Zugang die Möglichkeit zur Erweiterung auf Monitoringmechanismen. Enthalten sind neben frameworkartigen Möglichkeiten zu Verschaltung und Manipulation von Datenströmen auch Hilfsmittel z.B. zur Klassifikation von Musikstücken. Zusammen mit anderen Veröffentlichungen des Autors, vor allem zur Echtzeit-Visualisierung von Klassifikationsergebnissen [105], verspricht dieses System einen interessanten Ansatz. MARSYAS befindet sich allerdings noch in einem frühen Entwicklungszustand und verfügt nicht über Mittel, um Monitoring oder andere Visualisierungen performant zu gewährleisten.

Die vorliegende Arbeit erhebt zwar nicht den Anspruch, einen Standard schaffen zu wollen, sie möchte aber die Vorteile eines solchen noch zu schaffenden Standards aufzeigen. In diesem Sinne wird nachfolgend das im Rahmen der vorliegenden Diplomarbeit entwickelte GenMAD-Konzept vorgestellt, das eine flexible Verschaltung von Multimediadaten verarbeitenden Modulen verschiedenster Herkunft mit Mechanismen zur Handhabung und Visualisierung von Eventdaten anbietet. Im Gegensatz zu bisherigen Arbeiten auf diesem Gebiet erlaubt GenMAD die einfache Verwendung beliebiger akustischer Monitoring-Algorithmen über ein klar definiertes *Software Development Kit* (SDK) und stellt daher ein generisches Monitoringsystem für akustische Datenströme dar.

4.1.1 Wahl des Multimedia-Frameworks

Da GenMAD verschiedenartige Monitoring-Verfahren unterstützen soll, spielt die Wahl des ihm zugrunde liegenden Multimedia-Frameworks eine zentrale Rolle. Die maßgeblichen Entscheidungsgründe sind daher nachfolgend aufgeführt:

Das entwickelte Konzept basiert auf DirectShow, da nur dieses – im Gegensatz zu den anderen in Abschnitt 3.3 vorgestellten Multimedia-Frameworks – sowohl unter Visual Studio 6 kompilierbar ist als auch hinreichend tiefen Eingriff in das Framework erlaubt. Im Gegensatz zu einem vollständig eigenen Ansatz ohne ein zugrunde liegendes Multimedia-Framework bietet der verwendete Ansatz folgende Vorteile, vor allem in Anbetracht der begrenzt zur Verfügung stehenden Entwicklungszeit:

- Neben dem von Steinberg [123] entwickelten VST-Plugin-Format stellen DirectX-Plugins den zweiten wichtigen De-Facto-Standard unter Windows dar. Dies resultiert in einer großer Anzahl und Verschiedenheit existierender DirectX-Plugins, darunter hochqualitativen Filtern und andere Signalverarbeitungseffekten, die z.B. als Teil der Vorverarbeitungskette eines Identifikationsmoduls eine für eigene Entwicklungen nicht leistbare Qualität bieten können. Darüber hinaus ist auf diesem Weg auch ein problemloser Zugriff auf zukünftig entwickelte Algorithmen möglich. Hinzu kommen visuelle Analysemodule, die einen Echtzeit-Einblick z.B. in Form eines Spektrometers erlauben. Siehe z.B. [30] für eine Auswahl von DirectX-Filtern.

- DirectX besitzt eine durch seine zehnjährige Entwicklungsgeschichte, die enge Verzahnung mit dem Windows-Betriebssystem und den immensen Verbreitungsgrad begründeten Entwicklungsvorsprung hohe Performanz, die einhergeht mit in zahlreichen komplexen Projekten eingehend getesteter Stabilität.

- Das Aufsetzen auf den DirectX-Standard statt auf eine proprietäre Lösung führt zu weiteren Vorteilen: So wird, obwohl dies keineswegs Voraussetzung ist, die Einarbeitung bei vorhandenem Wissen um DirectX erleichtert, das stabile COM-Grundgerüst reduziert die Menge potentiell enthaltener Fehler um eine entscheidende Schicht und das kostenlos bereitgestellte SDK erlaubt, auch durch seine gute Dokumentation, eine leichte und tiefgehende Erweiterung.

- Über das Cedega-Projekt [24] ließe sich in Zukunft auch eine Portierung des Systems nach Linux gestalten, wobei der Funktionsumfang dies derzeit noch nicht erlaubt.

- Die unter anderem durch das WDM-Treibermodell in DirectX enthaltene Abstraktion von der Hardware-Ebene erlaubt den Zugriff auf spezielle Hardware-Fähigkeiten wie z.B. extern konfigurierbare Capturing-Treiber mit erweitertem Funktionsumfang, die über Plugins verfügbar gemacht wurden.

- Der durch DirectShow bereitgestellte große Funktionsumfang umfasst auch den Zugriff auf via Internet verfügbare Dateien, die durch die Angabe ihrer URL wie lokale Dateien transparent gestreamt werden können. Weitere Möglichkeiten betreffen sowohl die Konfiguration und die Automation von Plugin-Parametern als auch erweiterte Kontrollmechanismen wie etwa ein durch Interpolation der Quelldaten mögliches Echtzeit-Skalieren der Abspielgeschwindigkeit.

4.2 Das GenMAD-System

Das hier entwickelte *GenMAD*-System (Generisches Monitoringsystem für Akustische Datenströme) besteht wie in Abbildung 4.1 dargestellt aus zwei fundamentalen Komponenten, die über zwei Schnittstellen miteinander kommunizieren: Zum einen aus der Host-Applikation, zum anderen aus Filtern, die in einem Datenflussgraphen miteinander verbunden sind. Diese beiden Komponenten kommunizieren über Schnittstellen miteinander: Die GenMAD-Applikation implementiert das *IMonitoringMaster*-, Filter das *IMonitoringSlave*-Interface. Wie in der Abbildung gezeigt, kann der Graph auch Filter enthalten, die von Dritten entwickelt wurden und nur dem Signalfluss dienen. Diejenigen Filter, die das IMonitoringSlave-Interface implementieren und somit Monitoring-Funktionen unterstützen, sind – wie auch in der später erläuterten visuellen Oberfläche von GenMAD – durch weiße Schrift auf dunklem Hintergrund markiert; hier mit *B* und *C*. GenMAD kommuniziert darüber hinaus über separate Schnittstellen mit dem Datenflussgraphen und somit allen darin enthaltenen Filtern. Im Folgenden wird die Applikation kurz als *GenMAD* bezeichnet, das Gesamtsystem als *GenMAD-System*.

4.2.1 Grundlegendes Konzept des GenMAD-Systems

Wichtigstes Element von GenMAD ist der Datenflussgraph. Dieser ist ein gerichteter, azyklischer Graph und besteht aus einer Menge miteinander verbundener Module, genannt *Filter*, die verschiedene Signalverarbeitungsoperationen durchführen können.
In GenMAD werden zwei Typen von diskreten Signaldaten unterstützt. Zum einen Signalelemente d_{Audio} einer Audiodatenemenge $D_{\text{Audio}} := \{f \mid f\colon \mathbb{Z} \rightarrow \mathbb{R}\}$. Die Audiodatenelemente sind von der Form $d_{Audio}\colon \mathbb{Z} \rightarrow \mathbb{R}$, $d_{Audio} \subseteq D_{\text{Audio}}$, und ordnen einem Zeitpunkt einen reellen Wert zu. Die Differenz der Zeitpunkte zweier Audiodatenelemente entspricht stets dem Mehrfachen einer festen – als *Abtastrate* bezeichenten – Frequenz f_A. In GenMAD werden jedoch nur endliche Signale $f : [a : b] \rightarrow \mathbb{R}$, $b - a + 1 = N$, verarbeitet, die in sogenannten *Puffern* p_i der Länge $Len(p_i) := N$ gespeichert werden. Ein Puffer p_i ist ein Tupel aus einem einfachen Vektor und einem Zeitstempel: $p_i \in \mathbb{R}^N \times \mathbb{Z}$ mit einer Zeitdauer von N/f_A Sekunden. Der Zeitstempel eines Puffers wird über $Zeit : \mathbb{R}^N \times \mathbb{Z} \rightarrow \mathbb{Z}$ zurückgeliefert. Die Folge der Puffer wird als *Datenstrom* bezeichnet.

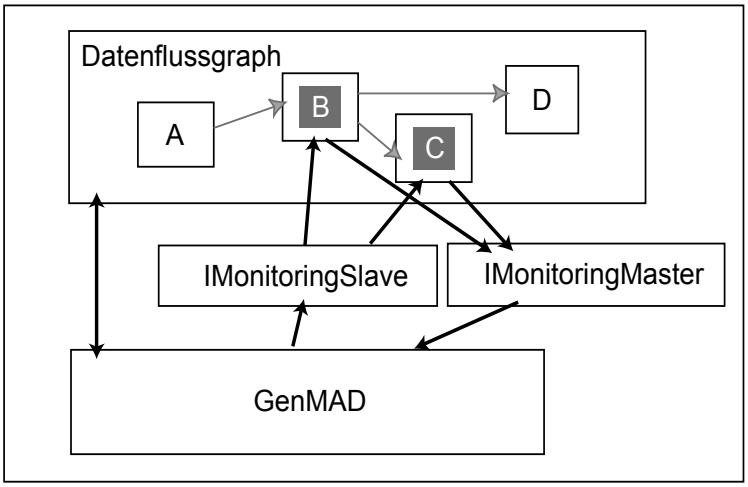


Abbildung 4.1: Schema des GenMAD-Systems

Daneben werden auch sogenannte *Eventdaten* d_{Event} einer Eventmenge $D_{\text{Event}} := \mathbb{Z} \times E$ unterstützt, die eine explizite Zeitangabe enthalten, $d_{Event} \subseteq D_{\text{Event}}$. E wiederum ist ein 10-Tupel, welches Informationen zu einem Ereignis enthält. Die Zeitangabe wird ebenfalls über $Zeit : \mathbb{Z} \times E \rightarrow \mathbb{Z}$ zurückgeliefert. Eventdatenelemente sind im Gegensatz zu Audiodaten zeitlich unabhängig voneinander. Trotzdem werden aufeinander folgende Eventdaten ebenfalls als *Datenstrom* bezeichnet, um eine einheitliche Beschreibung zu gewährleisten.

Um Aussagen zu ermöglichen, die wahlweise einen der beiden Datentypen betreffen, kann die folgende Menge verwendet werden: $DT = \{Audio, Event\}$.

Es sei darauf hingewiesen, dass in GenMAD verwendete Filter fremder Anbieter weitere Datentypen unterstützen können. Dieses können jedoch nicht von GenMAD-Filtern verarbeitet werden, sondern müssen ggf. in einen der beiden genannten Datentypen transformiert werden.

Das *i*-te GenMAD-Filter im Datenflussgraphen kann wie folgt modelliert werden:

$$F_i : (D_{Audio})^{k_A} \times (D_{Event})^{k_{E1}} \rightarrow (D_{Audio})^{k_A} \times (D_{Event})^{k_{E2}}$$

Dabei sind $k_A \in \{0, 1\}$ und k_{E1}, $k_{E2} \in \{0, 1, 2\}$ die (*Ein-* bzw. *Ausgangs-*) (*Audio-* respektive *Event-*)*Verbindungskardinalitäten*. Die *i*-te Eingangs- bzw. Ausgangs-Verbindungskardinalität vom Typ $T \in DT$ eines Filters F lässt sich wie folgt ermitteln: $Ein_{T,i}(F)$ bzw. $Aus_{T,i}(F)$. Die zweite Ausgangs-Event-*Verbindungskardinalität* des Filters F_2 beispielsweise ist $Aus_{Event,2}(F_2)$. Im Fall des Audiodatentyps kann der *i*-Index auch weggelassen werden, da die entsprechende Verbindungskardinalität k_A wie eben definiert höchtens eins ist. Die über alle Typen und Indizes kumulierte Ein- bzw. Ausgangs-Verbindungskardinalität eines Filters F wird mit $Ein(F)$ bzw. $Aus(F)$ bezeichnet. Entsprechend liefern $Ein_{Event}(F)$ bzw. $Aus_{Event}(F)$ die kumulierte Eingangs- bzw. Ausgangs-Verbindungskardinalität des Typs Event.

Bezeichne $F(d_A, d_E)_{Audio}$ die Projektion auf die Audiodatentyp-Komponente und $F(d_A, d_E)_{Event,i}$ die Projektion auf die *i*-te Eventdatentyp-Komponente der Ausgabe eines Filter F mit den Eingaben d_A, d_E. Ist die entsprechende Verbindungskardinalität $Aus_{T,i}(F) = 0$, dann gilt mit $T \in DT : F(d_A, d_E)_{T,i} = \{\}$.

Nun lässt sich die Ausgabe eines Filters F_1, das über einen Audiodatenstrom mit F_2 und über Eventdaten mit F_3 verbunden ist, folgendermaßen darstellen (wobei die Eingaben dem obigen Muster folgen):

$$F_1(F_2(d_{Audio1}, d_{Event1})_{Audio}, F_3(d_{Audio2}, d_{Event2})_{Event,i})$$

Auf diese Weise lassen sich auch komplexe Filter-Verschaltungen darstellen. Ist die Ausgabe eines Filters F_1 die Eingabe eines anderen Filters F_2 – wobei die Datentypen von Ein- und Ausgabe übereinstimmen müssen -, so nennt man die beiden Filter *verbunden* und spricht von einem *Datenfluss* oder *Datentransport* von F_1 zu F_2. Allerdings darf wegen der Kausalität des Graphen für jeden Index i jeden Typs $T \in DT$ die Ausgabe eines Filter nur Eingabe für einen einzigen Filter im Graph sein. Dies schließt insbesondere, da der Graph azyklisch ist, die Möglichkeit aus, dass die Ausgabe eines Filter direkt oder indirekt Teil der eigenen Eingabe ist.

Filter gibt es darüber hinaus in verschiedenen Typen. Die grundsätzlichste Unterteilung in drei Filterklassen leitet sich direkt aus den jeweiligen Ein- und Ausgangsverbindungs-Kardinalitäten ab. *Quellfilter* sind Filter mit $Ein(F) = 0$, *Rendererfilter* diejenigen, für die gilt: $Aus(F) = 0$. Als *Transformatorfilter* werden dagegen alle Filter F bezeichnet, für die $Ein(F) \geq 1$ *und* $Aus(F) \geq 1$ gelten. Ein Filter mit $Ein(F) + Aus(F) = 0$ ist nicht möglich.

Da Rendererfilter keine Ausgabe und Quellfilter keine Eingabe besitzen, ergibt sich ein Datenfluss von den im Graph vorhandenen Quell- über die Transformator- zu den Rendererfiltern. Ein Graph ist darüber hinaus erst dann *vollständig* und somit funktionsfähig, wenn sich mindestens ein Quell- und ein Rendererfilter darin befindet. Aufgrund ihrer Besonderheit sind es auch diese beiden Filtertypen, die den Datenfluss im Graphen grundsätzlich kontrollieren: Quellfilter initiieren den Fluss, während Rendererfilter ihn blockieren können. Daher setzt die durch den Benutzer beeinflussbare Datenflusskontrolle letztlich an genau diesen beiden Enden des Datenflussgraphen an.

Eng damit verwandt ist die Datensynchronität: Es muss gewährleistet sein, dass zu jedem Zeitpunkt $t \in \mathbb{Z}$ alle Datenelemente d mit $Zeit(d) \geq t$ an die Rendererfilter übergeben wurden. Dazu werden zwei Mechanismen kombiniert: Zum einen wird von (insbesondere Audiodaten verarbeitenden) Filtern erwartet, dass sie in Echtzeit und nach dem *FIFO*-Prinzip (*First In, First Out*) arbeiten, d.h. dass die Zeit zum Verarbeiten eines Puffers p durch das Filter F kleiner als die im Puffer enthaltene Zeitdauer ist, und dass eingehende Daten vollständig verarbeitet werden, bevor neue Daten angenommen werden. Zum anderen haben alle Filter Zugriff auf eine *Referenzuhr RefClock* $\in \mathbb{Z}$, um so Zeitstempel von Datenströmen mit der Referenzzeit vergleichen zu können.

Da die Ein- und Ausgangsverbindungskardinalität für Audiodaten höchtens eins ist, betrifft das Problem der Datensynchronisierung von Transformatorfiltern nur Filter mit einer Event-Eingangsverbindungskardinalität von zwei. Die Synchronisierung kann in diesem Fall über die Zeitstempel der Eventdatenelemente gewährleistet werden.

Aus der Definition der drei Grundfiltertypen ergibt sich die Möglichkeit, die miteinander verbundenen Filter des Datenflussgraphen als gerichteten Graphen darzustellen. Dabei wird jedes Filter durch einen Knoten repräsentiert. Quellfilter stellen die Wurzelknoten dar, Transformatorfilter die inneren Knoten und Rendererfilter die Blätter. Falls zwei Filter F_1, F_2 miteinander verbunden sind, wird dies durch eine gerichtete Kante zwischen den Filtern dargestellt, und an jedem der beiden Kantenenden wird der entsprechende Typ $T \in DT$ und ggf. der Index vermerkt, der für den Filter die Verbindung markiert. Ein solcher Beispielgraph ist in Abbildung 4.2 abgebildet.

Transformatorfilter lassen sich – bezogen auf ihren Einsatz in GenMAD – noch weiter typisieren, wobei in besonderen Fällen Filter mehreren Typen entsprechen können:

Gilt für ein Filter F und $T \in DT$: $Ein_T(F) > Aus_T(F) > 0$, so spricht man von einem *Kombinator*, im entgegengesetzten Fall $0 < Ein_T(F) < Aus_T(F)$ von einem *Splitter*. Gilt dagegen $Ein_{Event}(F) = Aus_{Event}(F)$ *und* $Ein_{Audio}(F) = Aus_{Audio}(F)$, so wird F *Verarbeitungsfilter* genannt. Für *Generatoren* gilt $0 = Ein_T(F) < Aus_T(F)$, für *Konsumenten* $Ein_T(F) > Aus_T(F) = 0$. Das in Abbildung 4.2 gezeigte Filter F_2 etwa ist nach dieser Definition ein Generator, F_5 dagegen ein Kombinator.

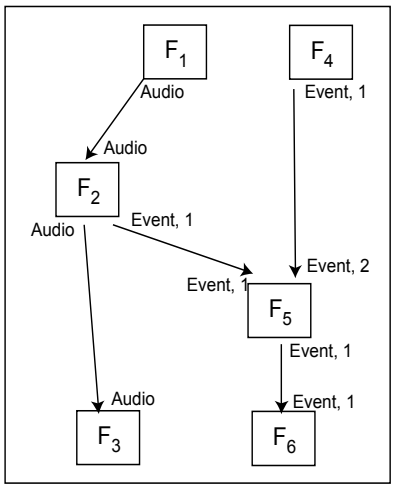


Abb. 4.2: Beispielgraph

Transformatorfilter können also vielfältige Aufgaben besitzen: Datenströme vereinen, verteilen, verändern, erzeugen oder zerstören. Erst durch diese Fähigkeiten wird ein maximal flexibles Arbeiten in GenMAD ermöglicht. Wie in Kapitel 5 erläutert, wurde im Rahmen dieser Arbeit eine Reihe verschiedener Transformatortypen konzipiert und umgesetzt.

Wie erwähnt, stellt neben der umfangreichen und flexiblen Manipulation des Datenflussgraphen die Datenvisualisierung eine zentrale GenMAD-Funktionalität dar. Da dies das Verwenden der Datenströme des Datenflussgraphen erfordert, wurde das Konzept des *Visualisierungsfilters* erstellt, das als Splitterfilter auch die GenMAD-Applikation zu einer Art abstraktem Rendererfilter macht und auf diese Weise eine echtzeitfähige und effiziente Visualisierung ermöglicht.

4.2.2 Zentrale Aspekte der GenMAD-Applikation

Primäre Ziele des GenMAD-Konzeptes sind:

- Bereitstellung einer grafischen Oberfläche zur visuellen Erstellung und Editierung eines Datenflussgraphen aus DirectX-Filtern
- Kommunikation mit und Konfiguration von Filtern
- Abspiel- und Transportkontrolle mit Fehlerbehandlung
- Projektzentriertes Arbeiten mit Möglichkeiten zum Laden/Speichern des aktuellen Datenflussgraphen
- Bereitstellung einer geeigneten, konfigurierbaren Echtzeit-Visualisierung von Audio- und Eventdaten mehrerer Filter

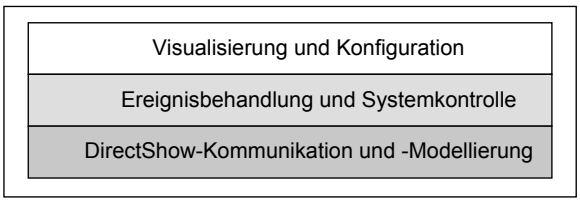


Abbildung 4.3: Schicht-Aufbau von GenMAD

Dazu werden die verschiedenen Aufgabenbereich möglichst klar in Klassen und diese abstrakt in Schichten getrennt. Dieser Schicht-Aufbau ist in Abbildung 4.3 dargestellt und erleichtert zum einen die Übersicht und somit die Erweiterbarkeit und das Debugging der Implementierung, zum anderen dient es durch das Ausführen in verschiedenen Threads und die Zugriffskontrolle (private, geschützte und öffentliche Methoden und Eigenschaften) der Robustheit bzw. Laufzeitsicherheit der Anwendung. Die Tiefe einer Schicht ergibt sich durch die Nähe zum DirectShow-Framework bzw. zu Applikations-basierenden Funktionen, etwa zur grundlegenden Ereignisbehandlung oder zur Erzeugung von Fensterobjekten. Dies geht nicht unbedingt einher mit einer abstrakteren Sicht (engl. *scope*) der Klassen, sondern eher mit einem abstrakteren Aufgabenbereich: Die enge Verzahnung mit DirectShow und das Ziel einer möglichst ressourcenschonenden und effizienten Anwendung impliziert zum einen Klassen, die sich über mehrere Schichten erstrecken und zum anderen auch in höheren Schichten wie der Visualisierung den Umgang auch mit nur rudimentär aufbereiteten Signaldaten.

In der tiefsten Schicht findet die direkte Kommunikation mit DirectShow-Komponenten statt. Die zentrale DirectShow-Komponente GraphBuilder wird ebenso durch eine Klasse gekapselt wie Filter und Pins. Um eine erweiterte Kommunikation mit den Filtern zu ermöglichen, ist in dieser Schicht auch die Implementation des IMonitoringMaster-Interfaces angesiedelt. Dies erlaubt eine geeignete Modellierung der von DirectShow bereitgestellten Funktionen zur Objekt-Erzeugung und –Kontrolle von Filtern und anderen Klassen und schafft somit die grundlegende GenMAD-Schicht, auf der eigene Funktionsschichten aufgesetzt werden können.

Auf dieser Schicht baut eine Ebene zur Behandlung von systemweiten Ereignissen und zur Kontrolle von Transportfunktionen auf. Diese stellen durch das Verwenden sowohl betriebssystemnaher als auch DirectShow-bezogener Funktionen tieferer Klassen und die Delegierung abstrakterer Klassen die Vermittlungsschicht von GenMAD dar.

Auf der obersten Ebene finden sich Klassen, die verschiedene Visualisierungssichten auf die beiden grundlegenden Signaltypen erlauben. Auch die visuelle Konfiguration und Anordnung von Filtern findet hier statt.

Die aufgabenspezifische Kommunikation von GenMAD mit den Filtern spielt eine zentrale Rolle im hier vorgestellten Konzept. Dies umfasst neben generellen Objekt-Kontrollmechanismen vor allem die über das IMonitoringMaster-Interface eingeführten Methoden, die nicht von DirectShow abgedeckte oder anwendungsspezifische Funktionalität bereitstellen. Dazu gehören folgende Funktionsbereiche:

- Abstimmung von Filter-Parametern im gesamten Graphen
- Push-Funktionen für Analyse und Visualisierung von Audio- und Eventdaten
- Mitteilung von internen Filter-Veränderungen
- Mitteilung von Datenstrom-Unterbrechungen
- Identifikation von Filtertypen

Zwar besitzt DirectShow eine eigene Infrastruktur für Ereignis-Mitteilungen, Funktionen mit hoher Datenlast überfordern jedoch die bereitgestellten Transportmechanismen. Diese Infrastruktur wird daher in GenMAD nur für wenige Funktionen verwendet, wie in der technischen Dokumentation beschrieben wird.

4.2.3 Filter

DirectX-Filter sind Windows-DLLs (dynamic link libraries), deren Basisklasse von der DirectShow-Klasse CBaseFilter abgeleitet ist. Dadurch wird eine grundlegende Behandlung durch das umgebende DirectShow-Framework ermöglicht, die z.B. die Erzeugung einer Instanz anhand eines identifizierenden COM-Bezeichners ermöglicht. Durch Ableiten weiterer DirectShow-Klassen und Überladen von Methoden sind entsprechende Anpassungen z.B. des lokalen Datentransports von Filter zu Filter möglich.

Durch optional implementierte DirectShow-Schnittstellen ist es möglich, Methoden zu definieren, deren Funktionsumfang über die vom DirectShow-System bereitgestellten minimalen Fähigkeiten hinausgeht. Dazu zählen z.B. Funktionen zur Neu-Positionierung von Zeigern eines Quellstroms, um bei der Wiedergabe einer Datei vor- oder zurückzuspulen, oder die Unterstützung von Speicher- bzw. Ladefunktionalität. Um eigene Funktionen bereitzustellen, können daher eigene COM-Interfaces unter Angabe einer GUID definiert und implementiert werden. Der GraphBuilder von DirectShow liefert unter Angabe dieser Interface-GUID einen Zeiger auf die so definierte Schnittstelle. Das entsprechend Abbildung 4.1 von GenMAD-Filtern unterstützte IMonitoringSlave-Interface soll die von DirectShow bereitgestellte vor dem gegebenen Anwendungshintergrund um zusätzliche Funktionen erweitern und ermöglicht so die in Kapitel 5 erläuterte Implementation der zu Beginn dieses Kapitels vorgegebenen Zielfunktionalität. Das IMonitoringSlave-Interface umfasst vor allem folgende Funktionsbereiche:

- Registrierung beim GenMAD-Host
- Methoden zur Festlegung und Abfrage der Blockpuffergröße
- Datentransport-bezogene Funktionen, z.B. Abfrage eines erweiterten Status
- Filter-Benennung
- Funktionalität zur Konfiguration durch den Benutzer
- Parameterabstimmung
- Visualisierungsfunktionen

Um den Aufwand zur Entwicklung eigener Filter so gering wie möglich zu halten, steht ein dafür konzipiertes und getestetes *Software Development Kit* (SDK) bereit. Dieses wird in Kapitel 5 erläutert und ist im Anhang technisch dokumentiert.

Kapitel 5

Implementation

5.1 GenMAD

GenMAD besitzt ein Hauptfenster, das aus einer Toolbar und einem Bereich zur visuellen Manipulation des Datenflussgraphen besteht und in Abbildung 5.1 dargestellt ist. Die fest im oberen Bereich positionierte Toolbar beinhaltet sowohl Buttons für Projekt-bezogene Operationen (Neues Projekt, Laden, Speichern) und für das Hinzufügen lokaler und im Internet verfügbarer Mediendateien als auch Buttons für das Hinzufügen neuer Filter sowie für den Aufruf des Visualisierungs-Fensters, die Kontrolle der Referenzuhr und des Transports. Darüber hinaus sind dort eine Anzeige der aktuellen Laufzeit des Graphen, ein sich analog zur ablaufenden Zeit bewegender Schieberegler für den manuellen Vor- und Rücklauf und ein editierbares Feld zur Eingabe eines Skalierungsfaktors über eine von DirectShow angebotene Interpolations-Methode enthalten.

Der Manipulationsbereich enthält die durch verschiedenartig kolorierte Rechtecke repräsentierten Filter des aktuellen Graphen mit namentlicher Beschriftung. Dabei sind Filter, die das IMonitoringSlave-Interface implementieren, dadurch markiert, dass ihr Name statt durch dunkle durch weiße Schrift auf dunklem Hintergrund hervorgehoben ist. Eingehende Pins sind als kleine graue Quadrate stets an der linken Seite jedes Filters, ausgehende an der rechten Seite befestigt und mit ihrem Namen versehen. Verbindungen zwischen Filtern respektive Pins sind durch Linien gekennzeichnet. Der Aufbau ist dabei inspiriert von dem im DirectX-SDK enthaltenen Programm *GraphEdit*. Sämtliche durch den Benutzer ausführbare Operationen sind entweder über die Toolbar erreichbar oder durch Editieren der genannten Objekte im Manipulationsbereich möglich. Per Rechtsklick kann ein Filter dialogbasiert konfiguriert werden. Darüber hinaus verfügen Filter und Pins zusätzlich über Kontextmenüs, die Informationen zu Zustand und Eigenschaften sowie verschiedene Operationen anbieten.

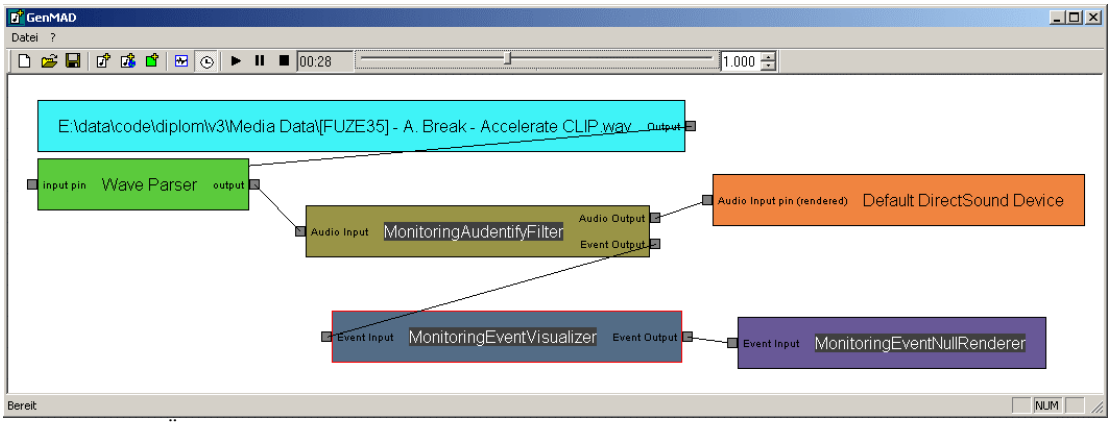


Abbildung 5.1: Überblick über das Hauptfenster von GenMAD

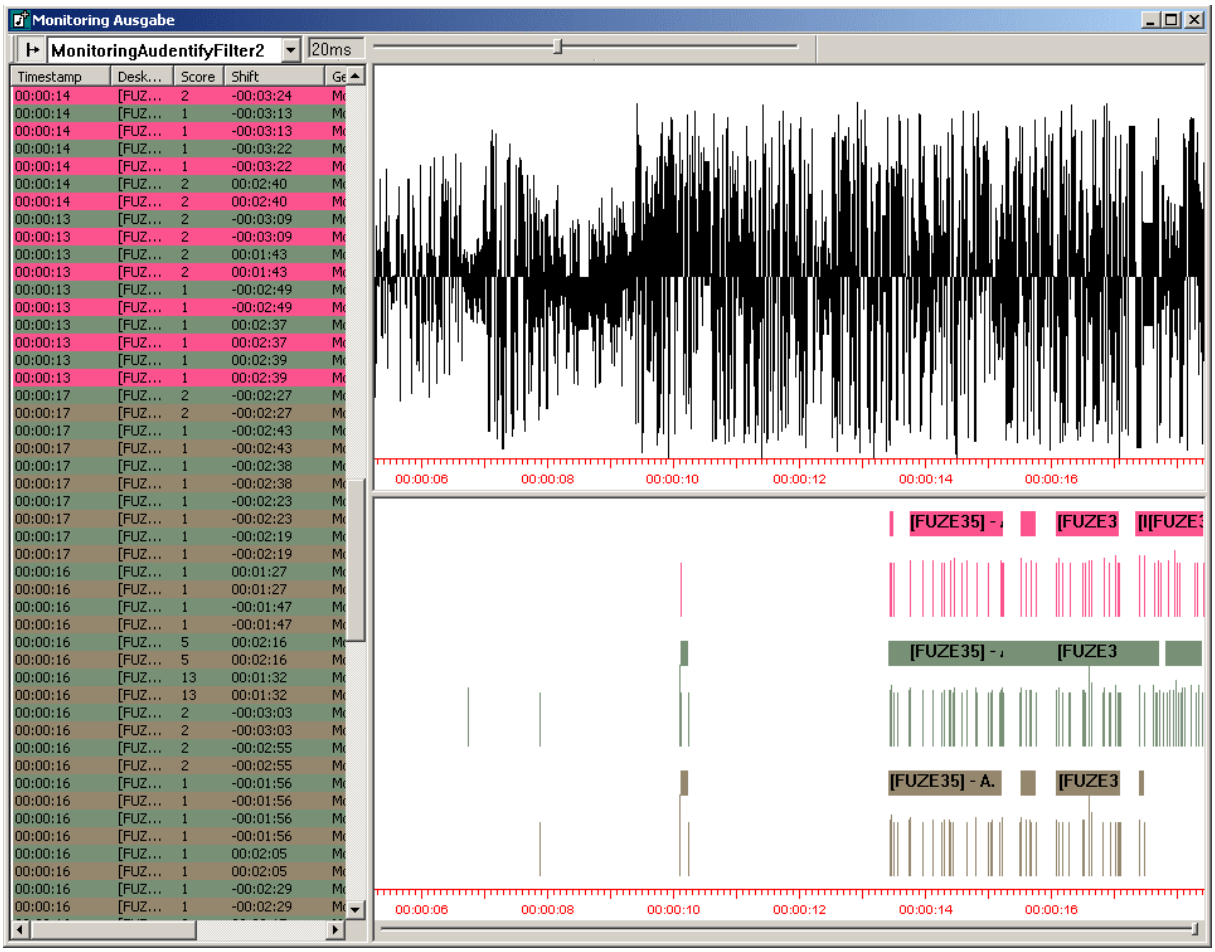


Abbildung 5.2: Das Visualisierungsfenster von GenMAD

Das in Abbildung 5.2 dargestellte Visualisierungsfenster ist zum einen ebenfalls mit einer Toolbar ausgestattet und zum anderen – entsprechend der in Abbildung 4.3 dargestellten drei Teilklassen zur Visualisierung – in drei Teilbereiche jeweils variabel einstellbarer Größe eingeteilt. Auf der linken Seite befindet sich eine in Listenform gehaltene Darstellung aller Events, die zu jedem Event auch die entsprechenden Parameter enthält. Die rechte Hälfte des Fensters enthält im oberen Teil eine Echtzeitdarstellung der Wellenform des ausgewählten Filters, falls der im folgenden erläuterte Audiofokus gesetzt ist, und im unteren eine Piano-Roll-ähnliche Echtzeit-Repräsentation der Events. Verschiedene Instanzen des zur Event-Visualisierung verwendeten *MonitoringEventVisualizer*-Filters werden übereinander angeordnet dargestellt. Die Farbgebung korrespondiert dabei, ebenso wie in der Liste, mit der Farbe der Filter im Manipulationsbereich. Jeder vertikale Strich der Eventdarstellung entspricht einem per Zeitstempel datierten Event, dessen Höhe seinem *Ranking* entspricht. Aufeinander folgende Events gleicher Quelle werden als *Eventblocks* visuell zusammenfasst, indem ein darüber liegender, mit dem Namen des Events beschrifteter horizontaler Balken gezeichnet wird, wie in Abbildung 5.2 ersichtlich.

Das erwähnte Konzept des Audiofokus bestimmt zu jeder Zeit maximal ein Filter, welches zum einen das IMonitoringSlave-Interface implementieren und zum anderen mindestens einen eingehenden Audio-Pin besitzen muss. Die Audiodaten des Filters mit dem Audiofokus werden im oberen rechten Fensterbereich in Echtzeit angezeigt. Die Auswahl kann außer der Wahl durch Doppelklick auf ein entsprechendes Filter im Manipulationsbereich auch durch die in der Toolbar enthaltene Auswahlbox erfolgen. Des weiteren enthält die Toolbar einen Button zum Umschalten der Echtzeit-Anzeigen auf *Standby* und einen beschrifteten Schieberegler zum Zoomen. Im Standby-Modus kann die Eventdarstellung per Schieberegler am unteren Fensterrand zeitlich durchlaufen werden und erlaubt so einen Rückblick.

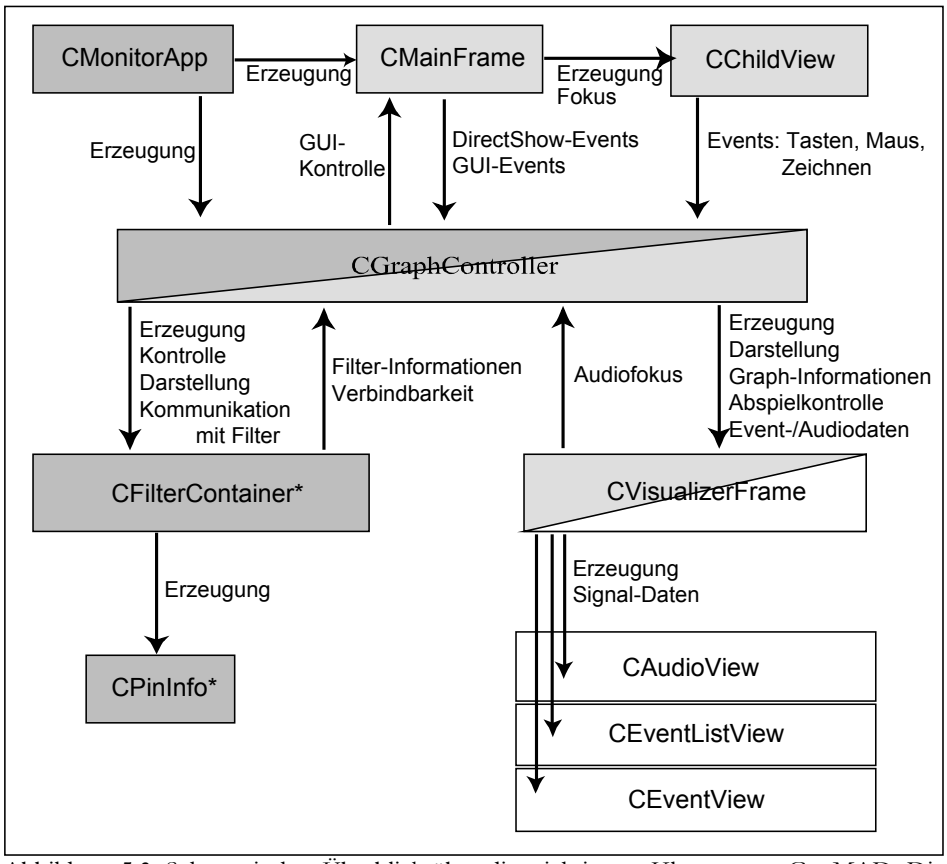


Abbildung 5.3: Schematischer Überblick über die wichtigsten Klassen von GenMAD. Die Farbgebung folgt dem in Abb. 4.3 eingeführten Schichtenmodell. Mit einem Stern (*) versehene Klassen werden nicht als Singletons verwendet.

Im Anhang befinden sich Hinweise zur Verwendung und Installation von GenMAD, sowie ein Benutzerhandbuch und die technische Dokumentation. Sämtliche im Rahmen dieser Diplomarbeit entwickelten Komponenten liegen als kompilierte Versionen und im Quelltext auf CD bei.

5.1.1 Zentrale Klassen und Strukturen von GenMAD

Die zentralen Klassen und die wichtigsten Beziehungen von GenMAD sind schematisch in Abbildung 5.3 dargestellt. Eine UML-Darstellung des gesamten Klassenbestandes ist in Anhang D.1 dargestellt. Im Folgenden sollen die Aufgabenbereiche der einzelnen abgebildeten Klassen kurz erläutert werden:

- **CMonitorApp:** Haupt-Applikationsklasse, die die Anwendung durch Erzeugen des Hauptfensters und der CGraphController-Klasse initialisiert;

- **CMainFrame:** Repräsentant des Hauptfensters, der sowohl GUI-Elemente zur Interaktion und zur Statusmeldung erzeugt und verwaltet als auch entsprechende Events und DirectShow-Events nach Vorverarbeitung an das CGraphController-Objekt weiterreicht;
- **CChildView:** Repräsentant des Zeichenbereiches, dient jedoch nur der Annahme von Benutzer- (Tastatur, Maus) und Zeichen-Events, die an CGraphController weitergeleitet werden;

- **CGraphController:** Ebenso wie der den Datenflussgraphen kontrollierende GraphBuilder die zentrale Komponente von DirectShow darstellt, ist diese ihn in GenMAD modellierende Klasse wichtigstes Element der Applikation. Sie enthält analog zum Graphen und damit verbundenen Funktionen z.B. zur Abspielkontrolle auch die Repräsentanten-Objekte der verwendeten Filter sowie Funktionen zu Eventbehandlung, Erzeugung und Manipulation von Graph und Filtern. Darüber hinaus implementiert sie das bereits beschriebene IMonitoringMaster-Interface und stellt dadurch die Kommunikationsschnittstelle der Filter dar. Dies impliziert auch die Initiierung der Signal-Visualisierung.

- **CFilterContainer:** Modelliert ein Filter und stellt durch Zugriff auf veröffentlichte DirectShow-Funktionen, die jeweils enthaltenen Pins und vor allem auf das IMonitoringSlave-Interface die Schnittstelle der Applikation zu jedem Filter dar.

- **CPinInfo:** Repräsentiert einen Pin;

- **CVisualizerFrame:** Klasse des Hauptfensters der Visualisierung, das die Organisation der untergeordneten Fenster-Teilbereiche übernimmt;

- **CAudioView:** Fensterklasse zur zoombaren Echtzeit-Visualisierung des Audiodatenstroms;

- **CEventView:** Fensterklasse zur zoom- und scrollbaren Echtzeit-Visualisierung des Eventdatenstroms mittels einer Piano-Roll-Ansicht;

- **CEventListView:** Fensterklasse zur Echtzeit-Auflistung des Eventdatenstroms;

Da Eventdaten eine zentrale Rolle in GenMAD spielen, soll ein kurzer Überblick über die Eigenschaften eines Events gegeben werden:

- Beschreibung des Events, z.B. Name eines identifizierten Musikstückes

- Zeitstempel

- *Score*: Ranking des Events

- *Shift*: zeitliches Mapping, z.B. auf das Auftreten des identifizierten Stückes in einer Datenbank

- *Generator*: Name des Filters, das den Event erzeugt hat

- Audiostrom-Informationen, z.B. Abtastrate und Kanalanzahl

- *Index*: fortlaufende Nummerierung

- *Visualisierer*: Name des Filters, das den Event zur Visualisierung übergeben hat

Nachfolgend werden wichtige Mechanismen von GenMAD erläutert.

5.1.2 DirectShow-Events

Events in DirectShow können sowohl von Filtern als auch durch den Datenflussgraphen versendet werden, z.B. zur Benachrichtigung über das Ende des Datenstroms oder bei abnormaler Beendung durch den Benutzer. Der verwendete Mechanismus folgt insofern dem in Windows gängigen Schema zum Versand von Nachrichten (engl. *messages*), als zum einen Events bis zu ihrer Verwendung in einer abfragbaren Warteschlange (engl. *queue*) aufbewahrt werden, und zum anderen ein Event einen numerischen Eventcode und zwei Parameter umfasst, die als 32 Bit-Integers entsprechend auch als Zeiger verwendet werden können. Der Zugriff erfolgt über verschiedene Schnittstellen: Filter verwenden das IMediaEventSink-Interface zum Versand, Applikationen die IMediaEvent- und IMediaEventEx-Schnittstellen zum Empfang. Um einen Event empfangen zu können, muss eine Applikation über das Vorhandensein eines Events in der Warteschlange informiert werden. GenMAD verwendet dazu ein *Window notification* genanntes Verfahren. Zunächst wird ein privater Eventcode definiert:

```
#define WM_GRAPHNOTIFY  WM_APP + 1
```

WM_APP ist der per Definition höchstwertige Eventcode, der von Windows versandt wird. Über das IMediaEventEx-Interface wird dieser Code DirectShow als derjenige EventCode mitgeteilt, der beim Eintreffen eines DirectShow-Events als normale Windows-Message an GenMAD versandt werden soll. GenMAD prüft beim Eintreffen dieses Codes die per IMediaEventEx-Interface ansprechbare DirectShow-Warteschleife. Der Vorteil dieses Verfahrens besteht darin, dass GenMAD ohnehin normale Windows-Messages behandelt und daher kein zusätzlicher Handler implementiert werden muss. Die zur Behandlung von DirectShow-Events nötige Mehrstufigkeit des Verfahrens ist allerdings der Grund, warum GenMAD zur zeitkritischen Kommunikation mit Filtern ein COM-Interface-gestütztes Verfahren verwendet.

5.1.3 Hinzufügen neuer Filter

Filter sind in DirectX nicht nur durch Angabe einer GUID – auch *CLSID* genannt, z.B. `{0x481de083, 0x4712, 0x4ab2, 0xab, 0x16, 0x8d, 0x80, 0x8e, 0x1c, 0xf9, 0xb5}` – eindeutig identifiziert, sondern darüber hinaus ebenfalls per GUID selbstständig einer Filter-Kategorie zugeordnet. Da nicht alle gängigen Kategorien in Zusammenhang mit GenMAD sinnvoll erscheinen, sind nur ausgewählte Kategorien in dem modalen Dialog enthalten, der dem Benutzer anhand einer Baumstruktur die im System registrierten Filter, sortiert nach genannten Kategorien, anbietet. Die Ermittlung dieser Filter erfolgt über ein *System Device Enumerator* genanntes COM-Objekt, das zunächst unter Angabe der entsprechenden GUID vom System erzeugt werden muss. In einer durch den Enumerator definierten Schleife werden von ihm sogenannte *IMoniker*-Instanzen zurückgeliefert, die das Auslesen des Filternamens über *PropertyBags* ermöglichen. Dieser wird unter verdecktem Speichern des aktuellen Schleifenindizes in der Baumstruktur veröffentlicht. Das Erzeugen eines durch den Benutzer bestimmten Filters verwendet einen beinahe identischen Algorithmus, der bis zu demjenigen Schleifenindex iteriert, der zum gewählten Filter im Baum gespeichert ist. Das IMoniker-Objekt erlaubt anschließend über die *BindToObject*-Methode das Füllen eines zuvor leeren Filter-Objektes. Da die PropertyBag auch das Auslesen der Filter-CLSID ermöglicht, wäre ein simpleres Verfahren zur Erzeugung des Filters anhand der CLSID wünschenswert. Diese Vorgehensweise schlägt jedoch leider im Fall von Capturing-Filtern aufgrund eines Bugs in der DirectShow-Implementation fehl.

Angewandt wird sie jedoch bei der Erzeugung der sogenannten *Filter-Favoriten*, die zum schnellen Zugriff ähnlich einer künstlichen Kategorie im Baum angezeigt werden und zum einen aus einigen vordefinierten, häufig verwendeten Filtern, zum anderen aus sämtlichen GenMAD-

Filter bestehen. Letztere werden dynamisch anhand des Namens-Präfix *Monitoring* eingelesen, wodurch auch zukünftig entworfene GenMAD-Filter unterstützt werden.

5.1.4 Speichern & Laden

Das aktuelle GenMAD-Projekt, bestehend aus Datenflussgraph und dem internen Zustand der Applikation, kann unter Verwendung eines CArchive-Objektes serialisiert und in einer Datei gespeichert werden. Damit eine Klasse diese MFC-Funktion verwenden kann, muss sie von CObject abgeleitet sein, je ein Makro in Header- und Implementationscode enthalten und die Funktion `void Serialize( CArchive& archive )` implementieren.

Die zunächst aufgerufene CGraphController-Klasse führt diese Operation auf allen enthaltenen Filtern aus. Diese speichern extern verfügbare Verbindungen, Pins und Zustand, darunter die Filter-CLSID. Falls ein Filter das *IPersistStream*-Interface implementiert, kann es selbstständig seinen privaten internen Zustand als *IStream*-Objekt zurückliefern. Dieses unterstützt eine Methode zur Speicherung des Inhaltes in einem IStorage-Objekt, welches eine komplette virtuelle Dateistruktur mit Verzeichnissen in einer einzelnen Datei verwaltet. Um das gesamte GenMAD-Projekt in nur einer Datei zu speichern, wird die von IStorage erzeugte Datei anschließend binär ausgelesen und dem seriellen CArchive-Strom hinzugefügt.

Entsprechend werden beim Laden nach einem Zurücksetzen des Graphen zunächst die Filter separat deserialisiert und in der `CFilterContainer::Recreate()`-Methode bei gegebener Unterstützung des *IPersistStream*-Interfaces durch Schreiben in eine temporäre Datei und das entsprechende IStorage-Objekt in den originalen internen Zustand versetzt. Quellfilter, die auf eine Mediendatei zugreifen, müssen damit neu initialisiert werden. Die abschließend ausgeführte `CGraphController::Recreate()`-Methode fordert ausgehend von den Quellfiltern jedes Filter dazu auf, zunächst all seine ausgehenden Pins erneut per DirectShow-Methodenaufruf zu verbinden und anschließend rekursiv auch nachgeschaltete (*downstream*) Filter dazu aufzufordern.

5.1.5 Hinzufügen von Mediendateien als Datenquelle

Eine vom Benutzer ausgewählte lokale oder per URL referenzierte Mediendatei wird durch die Methode CGraphController::RenderData() in den Graph eingefügt. Falls der Graph leer ist, wird der von DirectShow bereit gestellte Mechanismus zum automatischen Erzeugen eines kompletten, auf der Mediendatei und ihrem Format beruhenden Quellfilters verwendet. Anschließend werden alle Filter analysiert und in einer CList aus CFilterContainer-Objekten repräsentiert.

Falls sich bereits Filter im Graph befinden, wird ein zweistufiger Prozess eingeleitet, der zunächst einen temporären Graphen erstellt und diesen mittels des gerade genannten Verfahrens füllt. Um auf alle in GenMAD implementierten Funktionen zurückgreifen zu können, werden nun der ursprüngliche und der temporäre Graph vertauscht und der Vorgang ähnlich dem eines leeren Graphen durchgeführt. Nach dem Wiederherstellen des Originalzustandes werden die neuen Filter sukzessiv in den Graphen eingefügt und entsprechend verbunden. Analog zum Ladevorgang müssen auch hier besondere Maßnahmen im Fall von Quelldateifiltern ergriffen werden. Gleiches gilt für den Fall, dass ein Filtername bereits im Graphen vorhanden ist, denn manche Methoden des DirectShow-Graphen arbeiten auf den internen Filternamen und nicht auf den entsprechenden GUIDs. In einem solchen Fall wird versucht, vor dem Einfügen automatisch einen neuen internen Namen zu erzeugen. Diese internen Namen können allerdings nach der Instanziierung nicht mehr geändert werden. Die tatsächlich im Manipulationsbereich von GenMAD angezeigten Filternamen von GenMAD-Filtern sind davon allerdings unabhängig und somit unbetroffen und können über die Konfiguration des Filters editiert werden.

5.1.6 Filter-Konfiguration

Falls ein DirectShow-Filter die *ISpecifyPropertyPages*-Schnittstelle implementiert, kann seine Konfiguration über einen ein oder mehrere Seiten umfassenden modalen Dialog editiert werden. Die einzelnen Seiten können auch eine Grafik und entsprechende Ereignisbehandlungsmethoden beinhalten, um z.B. das Drehen an virtuellen Knöpfen zu ermöglichen. Der in CFilterContainer::ShowPropertyPage() implementierte Aufruf des Dialogs verwendet die Funktion `OleCreatePropertyFrame()`, die ein eigenständiges OLE-Objekt mit den genannten Eigenschaften erzeugt.

5.1.7 Zeit und Uhren

Jede Zeitangabe basiert in GenMAD auf einer kleinsten Zeiteinheit, die 100 Nanosekunden umfasst. Weiterhin werden Zeitangaben, z.B. Zeitstempel von Datenblöcken, üblicherweise relativ als *Streamtime* in Relation zu einer Startzeit des entsprechenden Streams angegeben. DirectShow verwendet in Form des GraphBuilder-Objektes eine sogenannte *Referenzuhr*, um alle Filter des Graphen zu synchronisieren. Diese wird z.B. durch einen Baustein auf der Soundkarte betrieben und über den entsprechenden Treiber bereitgestellt. Falls statt exakter Synchronität eine maximale Performanz erforderlich ist, kann die Referenzuhr über die Toolbar auf NULL gesetzt werden und ermöglicht dem Graphen so ein zeitungebundenes Arbeiten. In diesem Fall sollte jedoch – falls enthalten – der Audio-Renderer ein sogenannter *Null-Renderer* sein, der eingehende Audiodaten nicht an die Soundkarte weiterreicht, da sonst keine kontinuierliche Wiedergabe garantiert werden kann.

5.1.8 Datenfluss, Abspielkontrolle & Seeking

GenMAD bietet über das DirectShow-Interface *IMediaControl* die drei üblichen Operationen zum Abspielen an – Start, Pause und Stop. Die möglichen Übergänge der entsprechenden drei Graph- und auch Filterzustände sind dabei nur zwischen benachbarten Zuständen möglich, also Start ➔ Pause ➔ Stop ➔ Pause ➔ Start. Dabei senden Quellfilter im gängigen Push-Modus auch im Pause-Zustand Daten, die sie so über ihre Ausgangs-Pins lange an Eingangs-Pins nachgeschalteter Filter weiterschieben, bis diese blockieren. Genau das tun Renderer nach Erhalt eines Datenblockes im Pausemodus. Dies entspricht dem aus der Videotechnik bekannten „Standbild“, bei dem derselbe Frame ununterbrochen dargestellt wird. Sobald der Graph gestartet wird, hebt der Renderer die Blockade auf.

Bevor GenMAD durch die CGraphController-Klasse den Zustand des Graphen ändert, wird durch Aufruf der entsprechenden Methode jedes Filters zunächst überprüft, ob alle GenMAD-Filter bereit sind. Erst, wenn dies für alle Filter der Fall ist, wird versucht, in den *Run*-Modus zu schalten. Abschließend wird das Signal an verschiedene Objekte weitergegeben, darunter CVisualizerFrame.

Seeking, also der wahlfreie Vor- und Rücklauf, muss von allen im Graph enthaltenen Filtern in Form des *IMediaSeeking*-Interfaces unterstützt werden. GenMAD verwendet genau dieses Interface, um Neupositionierungen an den Graphen weiterzugeben. Dieser leitet die Anfrage an alle vorhandenen Renderer weiter, die, falls sie nicht über die entsprechende Funktionalität verfügen, rekursiv vorgeschaltete Filter anfragen, bis entweder der Befehl ausgeführt oder abgelehnt wird. Üblicherweise wird diese Funktionalität nur von Quellfiltern ausgeführt.

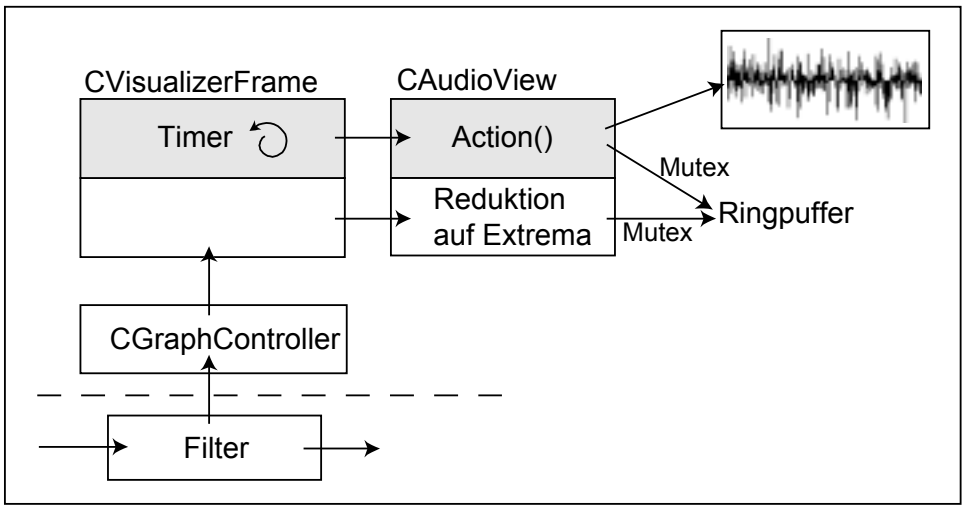


Abbildung 5.4: Visualisierung von Audiodaten. Verschiedene Threads sind farbig dargestellt.

5.1.9 Visualisierung

Das auf der CVisualizerFrame-Klasse beruhende Fenster zur Visualisierung wird dynamisch erzeugt und verfügt wie zu Beginn dieses Kapitels erläutert über drei Teilbereiche, jedes davon als von CWnd abgeleitete Klasse implementiert. Intern läuft ein Timer, der in Millisekunden-Abständen die drei Klassen zu Updaten auffordert und in einem zum CGraphController-Objekt separaten Thread ausgeführt wird. Bei Veränderung des Laufzeit-Zustandes, z.B. Start ➔ Pause, wird CVisualizerFrame analog dazu benachrichtigt und stoppt bzw. startet entsprechend den Timer.

Die zur Visualisierung benötigten Audio- und Eventdaten werden über separate Mechnismen erhoben: Demjenigen Filter, das den oben erläuterten Audiofokus erhält, wird dies per `IMonitoringSlave::SetShowAudioData()`-Methoden-Aufruf mitgeteilt, woraufhin es eingehende Audiodaten nicht nur an nachgeschaltete Filter, sondern auch an GenMAD weiterreicht. Eventdaten dagegen werden nur von einem Filtertyp, dem *MonitoringEventVisualizer*-Filter, an GenMAD weitergegeben.

Beide Datentypen werden dem CGraphController-Objekt über die IMonitoringMaster-Schnittstelle übergeben und direkt an die CVisualizerFrame-Instanz weitergereicht. Diese verteilt sie auf die drei Teilfenster:

Der CAudioView-Klasse wird ein *AudioBuffer*-Objekt übergeben, in dem sowohl ein Datenblock als auch Metadaten zu z.B. Abtastrate und Zeitstempel enthalten sind. Dieses wird mittels eines durch die aktuelle Zoomauflösung parametrisierten Algorithmus' in gleich große zeitliche Abschnitte zerteilt, zu denen je minimaler und maximaler Signalwert ermittelt und zusammen mit dem jeweiligen Zeitstempel in einem Ringpuffer abgelegt werden. Die vom CVisualizerFrame-Timer in kurzen Abständen aufgerufene Methode `CAudioView::Action()` berechnet das Zeitintervall seit dem letzten Aufruf und die entsprechend gewählter Zoomauflösung dafür benötigte Anzahl von Pixeln, wobei letztere zur Vermeidung von leeren oder doppelten Pixels als *double*-Werte verwendet werden. Zusammen mit dem Raster wird nun für jedes neue Pixel ein entsprechend skalierter vertikaler Strich gezeichnet. Es sei angemerkt, dass das Zeichnen zur Vermeidung von Flacker-Effekten nicht in dem tatsächlichen Fensterbereich, sondern über versteckte Puffer stattfindet, die anschließend vollständig und zur Fenstergröße skaliert in den Fensterbereich hineinkopiert werden. Um Probleme durch synchronen Zugriff zu vermeiden, greifen beide Threads unter Verwendung eines durch DirectShow bereitgestellten Mutex-Verfahrens seriell auf den Ringpuffer zu. Das Verfahren ist in Abbildung 5.4 dargestellt.

Das zur Anzeige von Eventdaten verwendete Verfahren ist ähnlich aufgebaut: Auch hier werden die vom Filter eingehenden Daten bis in die beiden verarbeitenden Klassen weitergeleitet

und dort in eine Speicherstruktur gefüllt. Allerdings handelt es sich hierbei um eine CList (eine doppelt verkettete Liste), und die Events werden, da sie aufgrund verschiedener Filterherkünfte möglicherweise in nichtsortierter Reihenfolge eintreffen können, zeitlich einsortiert. Darüber hinaus gibt es einen in diesem Zusammenhang aktualisierten Positionszeiger, der bei Eintreffen von Daten mit einem Zeitstempel kleiner als der bereits visualisierter Daten das Zeichnen auch dieser Daten garantiert; einmal gezeichnete Daten werden jedoch nicht noch einmal geschrieben. Die ebenfalls Zeichenpuffer-basierte, Timer-aufgerufene Zeichenmethode der CEventView-Klasse muss – im Gegensatz zum simplen Hinzufügen in eine Liste im CEventListView-Teilfenster – zusätzlich die Verteilung der verschiedenen Filter auf separate vertikal getrennte Bereiche, genannt *Bahnen*, dynamisch verwalten. Darüber hinaus werden auf jeder Filter-Bahn unterschiedliche Event-Beschreibungen ebenfalls dynamisch verschiedenen *Subbahnen* zugeordnet, die eine Filterbahn in vertikal getrennte Abschnitte unterteilen. Dazu bedient sich die Routine einer zentral in CVisualizer gehaltenen Liste aus *FilterDesc*-Objekten aller MonitoringEventVisualizer, die mit zusätzlichen Informationen z.B. zum vorigen Event und dem Beginn des aktuellen als zusammengehörend eingestuften Eventblocks versehen sind. Jedem Filter werden maximal 16 verschiedene Subbahnen zugestanden, die im Falle der Maximalauslastung bei Eintreffen bisher unbekannter Event-Beschreibungen dadurch neu strukturiert werden, dass die Subbahn der am längsten vergangenen Event-Beschreibung neu vergeben wird. Nach dem Zeichnen des Events wird geprüft, ob die Beschreibung des Event zu der des vorigen Events identisch ist und ob die Zeitdifferenz zu diesem vorigen Event einen definierten Grenzwert unterschreitet. In diesem Fall wird der aktuelle Eventblock verlängert und entsprechend eingezeichnet.

5.2 Das Filter-SDK

Um die Entwicklung neuer Filter zu vereinfachen, wurde im Rahmen der Diplomarbeit ein Software Development Kit (SDK) entworfen. Dieses beruht auf einer implementierten Basisklasse *CMonitoringBaseFilter* mit vier zusätzlichen Klassen für die verschiedenen Pintypen und benötigt nur wenige Angaben, um ein laufendes Filter zu erzeugen. Dadurch wird zum einen die Notwendigkeit zur Einarbeitung in DirectShow vollständig genommen, zum anderen führt die Fokussierung auf Monitoring-Aspekte zu einer erhöhten Produktivität bei der Entwicklung neuer Filter. Die dafür nötigen technischen Schritte sind in Anhang A aufgeführt.

Zur nötigen Registrierung eines Filters und dem späteren Erzeugen über eine *Factory*-Funktion benötigt Windows neben einer eindeutigen COM-CLSID und Informationen zu Filter-Namen und sonstigen –Eigenschaften vor allem auch explizite Angaben über Anzahl und Art der enthaltenen Pins. Diese werden in Form eines globalen Feldes `CFactoryTemplate g_Templates[]` erwartet, das auf andere Felder verweist, die alle benötigten Informationen enthalten.

Um dem Entwickler des SDK die Einarbeitung in diesen Mechanismus zu ersparen, enthält das SDK eine Datei namens `FilterDefs.h`, die auf übersichtliche Weise Präprozessor-Definitionen mit grundlegenden Angaben enthält, welche ein Entwickler anpassen muss. Im Fall des *MonitoringEventIncluder*-Filters sieht dies folgendermaßen aus:

```
//==================================================================
//Anzahl d. versch. Pintypen

#define PINS_AUDIO_IO 0
#define PINS_EVENT_IN 1
#define PINS_EVENT_OUT 1
```

```
//====================================================================
// *DER* GUID
const GUID IID_MonitoringEventIncluder = { 0x4c03cfaa, 0xb667, 0x43ee,
0x96, 0xaa, 0xa1, 0xea, 0x66, 0xfa, 0x6e, 0xed };

//====================================================================
//Definition d. Filter-Parameter für globale Makros
#define FILTER_CLSID          IID_MonitoringEventIncluder
#define FILTER_CLASS          CMonitoringEventIncluder
#define FILTER_NAME           "MonitoringEventIncluder"
#define FILTER_NAME_WIDE      L"MonitoringEventIncluder"

#define FILTERPROP_CLSID      CLSID_MonitoringBaseFilter_PropertyPage
#define FILTERPROP_CLASS      CMonitoringBaseFilterPropertyPage
```

In einer zweiten zentralen Datei des SDK namens `GlobalMacros.h` wird unter Zuhilfenahme dieser Definitionen das globale CFactoryTemplate-Feld automatisch durch Präprozessorangaben generiert und ermöglicht so eine korrekte Kompilierung des Filters. In der neuen, von *CMonitoringBaseFilter* ableitenden Filterklasse werden diese Angaben durch das Einbinden (per *Include*) der dritten wichtigen Datei `SDK.h` in Variablenwerte, z.B. Pin-Felder, geschrieben. Darüber hinaus werden so wichtige Funktionen parametrisiert. Zusätzlich zur Verwendung der *CMonitoringBaseFilter*-Klasse, in der das lauffähige Grundsystem zu z.B. Datentransport, Medienformatverhandlung und GenMAD-Funktionen enhalten ist, wird die neue Filterklasse also dynamisch mit aktuellen Angaben ausgestattet und stellt daher eher eine Art Grundstruktur dar, die ohne diese zentralen SDK-Dateien nicht kompilierbar wäre.

Im Folgenden sollen wichtige Teile der Filter-Implementation dargestellt werden, die in *CMonitoringBaseFilter* und den Pinklassen implementiert sind.

5.2.1 Medienformat-Abgleich

Das in DirectShow enthaltene Konzept zur Behandlung von Datenformaten besteht aus je einer Angabe zu Haupt- und Untertyp der blockweise verwendeten Daten. Beim Versuch, Ausrespektive Eingangs-Pins zweier Filter zu verbinden (engl. *hand-shaking*), muss sichergestellt werden, dass das nachgeschaltete Filter das angebotene Datenformat verarbeiten kann. Dazu rufen die Pins nach einem von DirectShow diktierten Schema verschiedene Methoden zum Vergleich und der Suche nach möglichen Datentransformationen von Formattypen auf. In GenMAD-Filtern wird eine solche Kontrolle dadurch sichergestellt, dass die Pin-Klassen jeweils zunächst selbstständig versuchen, ein akzeptables Format zu finden und erst im Anschluss die entsprechenden Methoden von *CMonitoringBaseFilter* aufrufen. Unterstützt werden sowohl unkomprimierte PCM-Audiodaten in 8 und 16 Bit Auflösung und bis zu zwei Kanälen sowie der selbst definierte Typ *MEDIATYPE_Event*.

5.2.2 Datenübertragung

Wie in Abschnitt 5.1.8 erläutert, verwenden alle GenMAD-Filter das Push-Modell zum Datentransport. Beim Erhalt eines neuen Datenblockes ruft daher der Eingangs-Audiopin die `PreProcessAudio()`-Methode des Filters auf. Dort wird zunächst eine Kopie der Daten an die GenMAD-Applikation übergeben, falls das Filter derzeit den Audiofokus besitzt. Anschließend wird der Datenblock an einen noch nicht vollständig gefüllten Datenpuffer mit einer vom Filter geforderten Größe angehängt und – falls dieser vollständig gefüllt ist – an die `ProcessAudio()`-Methode übergeben, wo die Daten von implementierten Filtern verwendet

werden können. Falls ein Ende des Datenstroms signalisiert wird, übergibt das Filter den offenen Puffer an diese Methode. Audiodaten werden automatisch an Eingangspins nachgeschalteter Filter weitergeleitet.

Anders ist dies bei Eventdaten: Diese müssen explizit über `SendEvent()` weitergegeben werden, wodurch die Implementation von Filterungs-Mechanismen ermöglicht wird. Auch für Eventdaten existiert der genannte zweistufige Verarbeitungsmechanismus, indem in `PreProcessEvent()` jedes als binärer Datenblock versandte Event in ein Objekt umgewandelt wird und nachfolgend in `ProcessEvent()` für High-Level-Verarbeitung zur Verfügung steht.

Der gesamte Vorgang findet innerhalb eines Filters im selben Thread statt. DirectShow erlaubt allerdings auch die Verwendung eines sogenannten Worker-Threads, der z.B. im Quellfilter *MonitoringXMLReader* eingesetzt wird.

5.2.3 Konfiguration

Um Filter-Entwicklern eine einheitliche und einfache Möglichkeit zur dialoggestützten Konfiguration zu geben, verfügt *CMonitoringBaseFilter* über einen Mechanismus, der die automatische Erzeugung eines Dialoges mit bis zu acht *Property* genannten Variablen ermöglicht. Diese Properties können über `AddProperty()` unter Angabe von anzuzeigendem Namen, Variablentyp und der Speicheradresse registriert werden. Der Filtername wird automatisch als erste Property registriert. Nach einer Änderung durch den Benutzer ist es möglich, auf Änderungen bestimmter Properties zu reagieren, in dem die Methode `SetProperty()` überschrieben wird. Intern werden diese Properties in einem Feld verwaltet, auf das von einer für den Dialog zuständigen Klasse `CMonitoringBase-FilterPropertyPage` zugegriffen wird. Diese füllt die in acht Zeilen angeordneten Benutzerelemente des in Abbildung 5.5 abgebildeten generischen Dialogs. Nicht verwendete Zeilen werden ausgeblendet. Die Eventhandler-basierte Verwaltung der Eingaben durch den Benutzer überschreibt anhand der Variablen-Zeiger deren vorhandene mit neuen Werten.

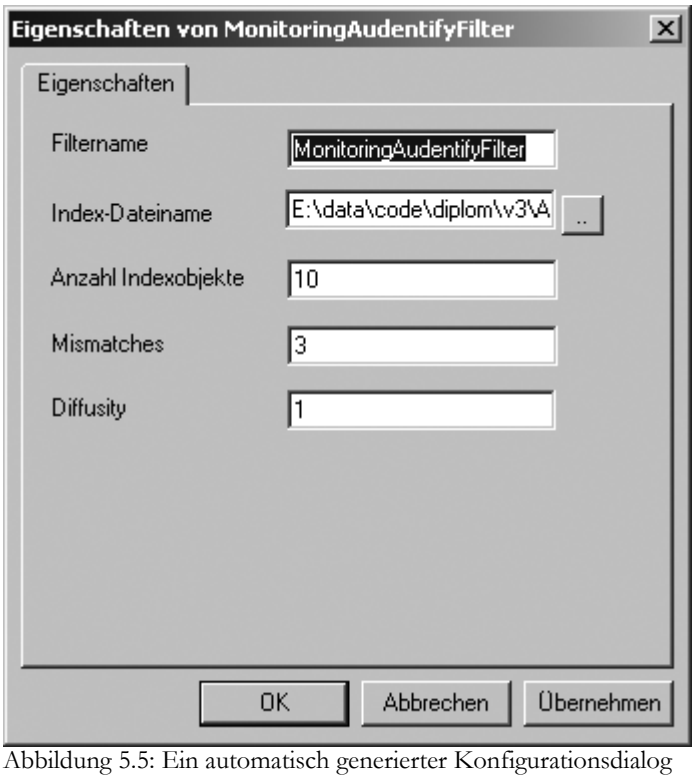


Abbildung 5.5: Ein automatisch generierter Konfigurationsdialog

5.2.4 Seeking

Um in einem Graphen Seeking-Fähigkeiten, also den wahlfreien Vor- und Rücklauf, verwenden zu können, muss jeder enthaltene Filter das IMediaSeeking-Interface implementieren. Seeking-Anfragen durch den GraphBuilder werden an die Renderer abgesetzt, die diese „stromaufwärts" weiterleiten bis die Anfrage bearbeitet wird oder ein Filter die Schnittstelle nicht implementiert. In CMonitoringBaseFilter werden sämtliche dieser Interface-Methoden dadurch implementiert, dass ankommende Anfragen direkt an ein vorgeschaltetes Filter weitergereicht werden, bevorzugt über solche, die über Audio-Pins verbunden sind.

5.2.5 Laden/Speichern

Die implementierten DirectShow-Schnittstellen eines Filters gibt dieses unter Angabe eines zum jeweiligen Interface entsprechend definierten GUID über die DirectShow-Methode `NonDelegatingQueryInterface` bekannt. Ist auch IPersistStream darunter, werden beim Laden und Speichern des Graphen auch interne Eigenschaften des Filters auf von diesem zu implementierende Weise behandelt. CMonitoringBaseFilter tut dies standardmäßig für alle registrierten Properties. Möchte ein GenMAD-Filter weitere Eigenschaften speichern, so muss es dies durch Überschreiben von `Write-` bzw. `ReadFromStream()` unter Verwendung eines IStream selbst implementieren.

5.3 Implementierte GenMAD-Filter

Die nachfolgend erläuterten Filter wurden im Rahmen dieser Diplomarbeit unter Verwendung des Filter-SDK entwickelt und getestet:

5.3.1 MonitoringAudentifyFilter

Dieses Generatorfilter verwendet den in der Arbeitsgruppe Multimedia-Signalverarbeitung entwickelten Audentify!-Algorithmus [57], der in Abschnitt 2.5.5 beschrieben wurde. Dieser ist als DLL eingebunden und benötigt als Parameter die Angabe der verwendeten Index-Datenbank. Diese ist ebenso als Property veröffentlicht wie weitere Algorithmus-spezifisch Parameter. Die DLL selbst ist statisch eingebunden und wird nach Parameteränderung informiert oder ggf. neu initialisiert. Nach Aufruf der zentralen Identifikationsroutine werden die gefundenen Treffer mit weiteren Informationen, z.B. zum Typ des Audiostreams und einem Zeitstempel, angereichert und als *EventSample*-Objekt verpackt an nachfolgende Filter versandt.

5.3.2 MonitoringEventNullRenderer

Dieser Renderer enthält keine eigene Logik, sondern dient nur mittels der in CMonitoringBaseFilter implementierten Methoden dem korrekten Abschluss einer Signalkette. Eingehende Events werden nicht weiterverarbeitet.

Das Filter erfüllt jedoch sehr wohl einen Sinn, da es als Renderer durch die implementierten DirectShow-Schnittstellen Anfragen des Graphen, z.B. zum Seeking, entgegen nimmt und an vorgeschaltete Filter weiterleitet. Erst dadurch wird ein vollständiges Funktionieren des Datenflussgraphen ermöglicht. Diese Aufgaben werden natürlich auch von anderen Filtern, wie z.B. dem in 5.3.8 beschriebenen MonitoringXMLWriter übernommen, diese führen jedoch noch zusätzliche Operationen aus und sind daher je nach Projekt ungeeignet.

5.3.3 MonitoringEventVisualizer

Dieses Filter dient der Visualisierung von Events in GenMAD. Es besitzt eine numerische Property namens „Score-Threshold“ mit Standardwert *1*, die bei eintreffenden Events mit dem darin enthaltenen Score-Ranking verglichen wird. Nur bei einem dem Threshold mindestens gleich hohen Score-Wert wird der Event an GenMAD übermittelt.

Das exklusive Verwenden dieses Filters zur Eventvisualisierung erlaubt eine viel gezieltere Auswahl visualisierter Filter, als wenn sämtliche Filter Events sendeten, denn in diesem Fall würde das GenMAD-Visualisierungsfenster durch Mengen redundanter Daten so unübersichtlich, dass ein gezieltes Arbeiten stark erschwert würde. Auch die automatische Visualisierung der *Event-Generatoren*, wie dem MonitoringAudentifyFilter, widerspräche dem hier gewählten flexiblen Ansatz, vor allem im Fall komplexerer Verschaltungen.

5.3.4 MonitoringEventSplitter

Dieses Splitterfilter dient der Verteilung von eingehenden Events an zwei nachgeschaltete Filter.

5.3.5 MonitoringEventMerger

Als einziges der hier vorgestellten Filter besitzt dieses Kombinatorfilter zwei eingehende Event-Pins. Zweck ist das Zusammenführen zweier Event-Pipelines unter Verwendung eines Algorithmus', welcher Ereignisse bewertet und ggf. verwirft. Diese Bewertung kann durch den Benutzer mittels mehrerer Parameters gesteuert werden:

Zum einen gewichtet das Filter Ereignisse auf den beiden Eingängen separat mit je einem Faktor. Der Parameter *Diffusion* steuert in Abhängigkeit der Zeitstempel vergangener Ereignisse den Werteverfall jedes Ereignistyps und erlaubt so eine zeitbasierte dynamische Bewertung. Schließlich wird nur dann ein Ereignis weitergeleitet, wenn innerhalb eines parametrisierbaren Zeitintervalls in der unmittelbaren Vergangenheit nur dieser Ereignistyp versendet wurde. Das Filter erlaubt auf diese Weise das Verhindern von „Ausreißern“ oder anderen widersprüchlichen Events und somit eine Segmentierung des Ereignisstroms. Da dieses Filter eine wichtige Rolle im Kapitel 6 spielt, soll dieser Mechanismus etwas erläutert werden.

Zur Kombination von Eventströmen hält das Filter in einem Feld `events` fixer Größe die Beschreibungen der letzten eingegangenen Events, unterschieden nach Event-Beschreibung. Trifft ein darin nicht enthaltener Event `es` ein, so wird dieser zum Feld an Stelle `iEvent` hinzugefügt, wobei im Fall der Maximalauslastung des Feldes der am längsten vergangene Event überschrieben wird. Nun wird die Bewertung unter Verwendung einer *Winner-Takes-All*-Strategie mit Gewinner-Feldindex `iCurrentWinner` durchgeführt:

```
//berechne vergangenen wert
REFERENCE_TIME rtDiff = es.rtTimestamp - events[iEvent].rtLastOccurance;

//bisheriger Wert des Events mit zeitlichem Verfall
double dValue = ValueFunction(events[iEvent].dLastValue, rtDiff);

//berechne daraus den aktuellen Wert
dValue += (double)es.lScore;

//bisheriges Maximum überschritten?
bool bNewWinner = (iCurrentWinner == -1);
if (iCurrentWinner >= 0)
{
   //selber Gewinner wie zuletzt?
   if (iCurrentWinner == iEvent)
```

```
   {
      bNewWinner = true;
      rtCurrentSegmentLength += rtDiff;   //Segment geht weiter
   }
   else
   {
      //Berechne Wert des bisherigen Gewinners
      double dWinnerValue = ValueFunction(
                  events[iCurrentWinner].dLastValue,
                  es.rtTimestamp - events[iCurrentWinner].rtLastOccurance);

      //bisherige + halber abstand zu vorigem gewinner-event
      if ((rtCurrentSegmentLength +
          (es.rtTimestamp - events[iCurrentWinner].rtLastOccurance) / 2)
          >= iMinimumSegmentLength*1000*10)
      {
      //neue Segmentlänge ist halber Abstand zu vorherigem gewinner-event
         rtCurrentSegmentLength = (es.rtTimestamp -
                           events[iCurrentWinner].rtLastOccurance) / 2;
         bNewWinner = true;
      }
      else
         //aktuelles segment fortsetzen
         rtCurrentSegmentLength += rtDiff;
   }
}
```

Falls ein Gewinner gefunden wurde, wird dies vermerkt und der Event gesendet. Die Funktion `ValueFunction()` berechnet dabei den durch die Diffusions-Property parametrisierten linearen zeitlichen Verfall der Bewertung. Entsprechend findet bei einem Diffusionswert von *0* keine Filterung, sondern ein reines Durchleiten statt:

```
double CMonitoringEventMerger::ValueFunction(double dValue, REFERENCE_TIME
rtDiff)
{
      //umrechnen in Referenz-Zeit (100ns Auflösung)
      REFERENCE_TIME rtDiffusion = iMSecDiffusion*1000*10;
      //anwenden
      dValue -= (rtDiff/rtDiffusion);
      //ggf. abschneiden
      return max(0, dValue);
}
```

Das Filter arbeitet darüber hinaus auch bei nur einer eingehenden Verbindung. Dies kann z.B. zur Filterung eines Eventstromes eingesetzt werden.

5.3.6 MonitoringEventExcluder

Dieses Verarbeitungsfilter verwendet eine *Blacklist* von Event-Beschreibungen, um Events zu filtern. Dazu stehen per Konfigurationsdialog bis zu acht Felder zur Verfügung.

5.3.7 MonitoringEventIncluder

Identisch mit MonitoringEventExcluder, allerdings wird hier eine *Whitelist* verwendet, so dass nur darin enthaltene Events weitergeleitet werden. Diese beiden Filter ermöglichen ein effektives, gezieltes Filtern von Events, z.B. im Szenario, dass GenMAD das Auftreten nur ganz bestimmter Events überwachen und alle anderen Events ignorieren soll.

5.3.8 MonitoringXMLWriter

Bei diesem Filter handelt es sich um ein Renderer-Filter, das den eingehenden Eventstrom in eine per Property auszuwählende XML-Datei schreibt. Da Events nicht garantiert in der korrekten zeitlichen Reihenfolge eintreffen, werden sie bis zum durch Stoppen des Graphen ausgelösten Schreiben in einer sortierten Liste gehalten. Die XML-Datei folgt dabei folgender DTD:

```
<?xml version="1.0"?>
<!DOCTYPE event [

   <!ELEMENT event (index, timestamp, description, shift, score, generator,
             sample_rate, bytes_per_sample, channels)>
      <!ELEMENT index (#PCDATA)>
      <!ELEMENT timestamp (#PCDATA)>
      <!ELEMENT description (#PCDATA)>
      <!ELEMENT shift (#PCDATA)>
      <!ELEMENT score (#PCDATA)>
      <!ELEMENT generator (#PCDATA)>
      <!ELEMENT sample_rate (#PCDATA)>
      <!ELEMENT bytes_per_sample (#PCDATA)>
      <!ELEMENT channels (#PCDATA)>
]>
```

5.3.9 MonitoringXMLReader

Dieses Filter ist das einzige implementierte Quellfilter. Es liest Events aus einer XML-Datei mit dem im vorigen Abschnitt definierten Format und sendet diese an Filter stromabwärts.

Um ein Reagieren auf Graphen-Befehle in Echtzeit zu ermöglichen, verwendet das Filter einen sogenannten *Worker-Thread*, der in einem separaten Thread das eigentliche Auslesen und Senden übernimmt. (Im Folgenden wird der Haupt-Thread des Filters einfach als *Filter* bezeichnet.) Der Worker läuft in einer durch das Filter terminierbaren Schleife und wartet auf Transportbefehle wie Start und Stop, die das Filter nach Erhalt durch den Flussgraphen ebenso weiterreicht wie Befehle zum Seeking.

Der Worker startet, da er anders als viele andere Quellfilter im Push-Modus arbeitet, auf den Start-Befehl hin eine verschachtelte zweite Schleife. In dieser werden entweder (mehrere) neue Events aus der Datei eingelesen oder unter Verwendung der Referenzuhr des Graphen alle Events versendet, deren Zeitstempel kleiner oder gleich der aktuellen Zeit ist. Zum Lesen der Datei wird eine ereignisbasierte XML-Komponente verwendet, die auf der Grundkomponente von David Hubbard [49] beruht und um eigene Anforderungen erweitert wurde. Der Lesevorgang besteht aus den folgenden schematisch dargestellten sukzessiven Schritten:

1) Öffnen der Datei
2) Testverfahren für Mindestlänge und enthaltene Event-Tags
3) Ermitteln des Zeitstempels des letztes enthaltenen Events
4) Ausführen einer Seek-Operation
5) Lesen von maximal 65kB Daten

6) Schließen der Datei
7) Übergabe der Daten an den XML-Parser

Falls der Parser Events findet, werden diese in *EventSampleObject*-Objekte verpackt und zu einem Vektor hinzugefügt, der von der Sende-Schleife abgearbeitet wird.

Vor- und Rücklauf werden über das Versetzen eines Positionszeigers in der Datei ausgeführt. Da die Events in einer Datei eine Start- und eine Endposition besitzen und im Allgemeinen nicht linear über ihre Zeitstempel in der Datei verteilt sind, wird ein rekursives Schätzverfahren angewandt. Es wird versucht die Dateiposition unmittelbar vor demjenigen Event zu finden, der den bezüglich des vom Graphen angeforderten nächstgrößeren Zeitstempel besitzt. Das Schätzverfahren versucht anhand bekannter Zeitstempel bekannter Events auf die Position in der Datei zu schätzen und schiebt dabei zwei eingrenzende Positionierungs-Zeiger sukzessiv auf einander zu, wie im nachfolgend etwas vereinfachten Algorithmus dargestellt:

```
HRESULT CWorker::SeekPos(REFERENCE_TIME rtNewPos, FILE *pFile, long
                                                  lEstSeekPos, XmlStream xml)
{
   //rtNewPos: angeforderter Zeitstempel
   //lEstSeekPos: aktuell geschätzte Dateiposition

   //geschätzte Position in Datei (mit Länge lXMLSize)
   lEstSeekPos = min(lEstSeekPos, lXMLSize-64);

   //suche den dort befindlichen Event per Hilfs-Funktion
   if (!FindEventAtPos(lEstSeekPos, pFile, xml))
      return E_FAIL;

   //check den gefundenen Event: exakt übereinstimmend?
   if (curEvent.rtTimestamp == rtNewPos)
      {iCurReadPos = lEstSeekPos; return S_OK;}

   if (curEvent.rtTimestamp < rtNewPos)
   {
      //zu früh: Variablen setzen -> Ziel eingrenzen
      lPosBefore = lEstSeekPos; indexBefore = curEvent.index;
      rtBefore = curEvent.rtTimestamp;
   }
   else
      [..] //zu spät -> das Gleiche für die "After"-Variablen

   //falls before- und after-Index aufeinanderfolgen -> gefunden!
   if ((indexBefore+1 == indexAfter) || (indexAfter == 0))
      {iCurReadPos = lPosAfter;     return S_OK;}
   else
   {
      //ggf. rekursiv vor- oder rückwärts weitersuchen, dazu neue Schätzung
      long lNewEstSeekPos = 0;

      //Was liegt weiter vom Ziel weg? Before- oder After-Pointer?
      REFERENCE_TIME rtDifBefore = rtNewPos-rtBefore;
      REFERENCE_TIME rtDifAfter = rtAfter-rtNewPos;

      // den weiter entfernten Zeiger (in gleichem Verhältnis wie den
      // anderen) ans vermeintliche Ziel heranschieben
      if (rtDifBefore > rtDifAfter)
      {
         //entweder neu anteilig berechnen oder zumindest minimal weiter!
         if (lPosBefore < lEstSeekPos)
            lNewEstSeekPos = lPosBefore + (lEstSeekPos-lPosBefore)/2;
         else
```

```
            lNewEstSeekPos = max((lPosAfter - lPosBefore)/2,
                                 lPosBefore+256);
      }
      else
         [..] //entsprechend für den umgekehrten Fall

      //Rekursion ausführen
      return SeekPos(rtNewPos, pFile, lNewEstSeekPos, xml);
   }
   return S_OK;
}
```

Die Funktion `FindEventAtPos()` verwendet dabei den XML-Parser, um in demjenigen Bereich der Datei nach einem Event zu suchen, der sich ab der übergebenen Position erstreckt.

5.3.10 MonitoringClassifier

Dieses Generatorfilter verwendet eine im Rahmen dieser Diplomarbeit konzipierte und realisierte Methode zur Klassifikation eingehender Audiosignale in Stille, Sprache oder Musik. Zur Diskriminierung der eingehenden Signale werden die folgenden Audiomerkmale verwendet:

- Varianz der Zero-Crossing-Rate
- Low-Energy-Frame-Percentage
- Varianz des Spectral Flux
- Varianz des Spectral Centroid
- Varianz des Spectal Roll-Off

Diese Merkmale wurden nach in verschiedenen Veröffentlichungen [108, 93, 34] publizierten Angaben zu Güte und Geschwindigkeit ausgewählt.

Die Werte werden aus Hamming-Frames mit einer Fensterlänge von 512 Samples und einer Überlappung von 384 Samples gewonnen. Varianzen beruhen auf den in einem Ringpuffer gehaltenen Framewerten der letzten Sekunde. Die Diskriminierung in die genannten drei Klassen erfolgt durch ein zweistufiges Verfahren: Zunächst wird das Signal anhand eines Energiemaßes auf Stille untersucht. Zwar wurden auch andere Verfahren, wie unter anderem das von Panagiotakis & Tziritas [82] beschriebene Verfahren untersucht, die Klassifikationsleistung war jedoch bei dem Energiemaß am höchsten. Dieses vergleicht den berechneten Wert mit einem konfigurierbaren Schwellwert. Dessen Standardwert wurde empirisch ermittelt, in dem eine Auswahl von als Stille deklarierten Audiosignalen untersucht wurde.

Falls keine Stille klassifiziert wurde, wird ein auf Trainingsdaten arbeitendes k-NN-Verfahren zur Klassifikation in Sprache und Musik eingesetzt, welches sich aus Gründen der Performanz eines k-d-Baumes bedient, um darin die trainierten Merkmale zu halten. k-d-Bäume sind Verwandte klassischer binärer Bäume, die – wie in der grundlegenden Arbeit von Friedman, Bentley und Finkel [39] beschrieben – *d*-dimensionale Daten in Zeit $O(n \log n)$ in achsenparallelen Hyperebenen ablegen können, wobei $O(n)$ Platz benötigt wird. Dazu werden die Daten auf jeder Ebene anhand der Informationen nur einer Dimension in zwei disjunkte Teilmengen aufgeteilt. Die erwartete benötigte Zeit für die Suche im Baum beträgt lediglich $O(\log n)$ und ermöglicht daher das effiziente Ablegen sehr großer Vektorenmengen, wie in Kapitel 6 aufgezeigt.

Zu jedem zu klassifizierenden Merkmalsvektor werden die im Baum befindlichen k nächsten Vektoren gefunden und in üblicher Weise anhand eines Votings zu einer Entscheidung

herangezogen. Inspiriert von dem von El-Maleh et al. [34] vorgeschlagenen Verfahren werden anschließend vorige Klassifikationen zu einer Mehrheitsentscheidung verwendet, um das Auftreten von Ausreißern zu minimieren. Im Gegensatz zu dem von El-Maleh vorgeschlagenen Verfahren, das nur drei Klassifikationen in das Voting mit einbezieht, arbeitet die hier verwendete Methode auf den Ergebnissen der letzten 250ms. Ein positiver Nebeneffekt ist die dadurch deutlich reduzierte Anzahl versendeter Klassifikationsevents.

Die Konfigurations-Parameter und verschiedenen Modi des Filters für das Trainieren und Erkennen von Signalen sind in Anhang F beschrieben.

5.3.11 Weitere filterbasierte Problemlösungen

Zwei weitere zu Beginn des 4. Kapitels genannte Ziele lassen sich über bereits bestehende Filter lösen:

- Zum einen erlauben Capturing-Quellfilter, die üblicherweise als Teil der Treiberarchitektur eines Soundchips installiert werden und sich in Ausstattung und Parametern zum Teil stark unterscheiden, das Verwenden von in Echtzeit in den Rechner eingehenden Online-Audiodaten. In einem Graphen lassen sich solche Filter wie gewöhnliche Datei-Quellfilter verwenden. Allerdings können aus offensichtlichen Gründen keine Seeking-Funktionen verwendet werden. Auch andere zeitbasierte Mechanismen können je nach Capturing-Filter eingeschränkt sein.
- Auch das Mitschneiden von Audiosignaldaten ist durch eine Kombination zwei Filter möglich. Dazu wird ein *Wavedest* genanntes Filter verwendet, das Teil des DirectX9-SDK ist und intern die für WAV-Dateien nötige Formatierung zu den reinen Datenblöcken hinzufügt. Der Ausgangspin dieses Transform-Filters muss mit einem „File Writer“-Filter verbunden werden, welches standardmäßig mit DirectX installiert wird. Der Pfad der Zieldatei kann dabei frei gewählt werden.

Kapitel 6

Anwendungen, Tests und Bewertungen

6.1 Test des Klassifikationsfilters

Um verschiedene Eigenschaften des in Abschnitt 5.3.10 erläuterten MonitoringClassifier-Filters zu evaluieren, wurde eine Serie mehrerer Tests durchgeführt, die nachfolgend beschrieben sind. Dazu wurde zunächst der Einfluss verschiedener Parameterwerte und Verschaltungsmuster auf die Klassifikationsleistung untersucht, bevor in zwei abschließenden Tests die korrekte Klassifikation von zwei längeren Radiomitschnitten mittels eines komplexeren Projektes getestet wurde.

6.1.1 Datenauswahl

Training und Parametertests wurden anhand von zwei Mengen aus Signaldaten durchgeführt, wobei jede Menge aus ca. 15 Sekunden langen Audioelementen eines Datentyps besteht und insgesamt je 20 Minuten umfasst. Zusätzlich wurde ein 90 Sekunden langes Stück aus verschiedenen, sehr leisen Signalen verwendet, um den Standardwert des im Filter enthaltenen separaten Stille-Erkennungsmoduls zu ermitteln. Wichtiges Ziel bei der Auswahl der Trainingsdaten war es, ein möglichst weites Spektrum verschiedener Musikgenres und Stimmen miteinzubeziehen, insbesondere weil der Klassifikator anhand von Radiomitschnitten getestet werden sollte. Die Daten entstammen daher sowohl FM-Radioaufnahmen, CDs als auch MP3s in unterschiedlichsten Qualitätsstufen und enthalten neben populärer Musik und Rock auch Klassik, Jazz, Ambient, elektronische Dancemusic und Schlager. Ähnlich sorgfältig wurden männliche wie weibliche Stimmen in verschiedenen Stimmungen und Lautstärken verwendet, wobei diese teilweise mit Hintergrundrauschen unterlegt sind oder einem telefonartigen Effekt unterliegen. Neben deutschen Stimmdaten sind weiterhin auch englische, französische, niederländische und japanische Elemente enthalten. Um die auch für Menschen oft schwierige Unterteilung in Musik oder Sprache zu unterstützen, wurden auch rein vokale Versionen von Popstücken als Sprache beschriftet.

Die Trainingsdaten sind dabei mit einer Abtastrate von 44,1 kHz und in 16 Bit Qualität abgespeichert. Wurden für einen Test Signale in niedrigerer Qualität benötigt, so wurden diese mittels der Software CoolEdit heruntergerechnet.

6.1.2 Audiomerkmale

Vier der fünf verwendeten Audiomerkmale (*Zero-Crossing-Rate*, *Spectral Flux*, *Spectral Centroid* und *Spectral Rolloff*) arbeiten nicht auf den eigentlich berechneten Werten, sondern den daraus abgeleiteten Varianzwerten, und auch das *Low-Energy-Percentage*-Merkmal greift auf laufende RMS-Mittelwerte zurück. Da zur Berechnung von Mittelwerten und Varianzen die Werte der vergangenen Sekunde herangezogen werden, haben alle Merkmale eine *Latenz* genannte Verzögerung gegenüber den Audiodaten von einer Sekunde. Aus Gründen der Performanz wurden die zur Spektraltransformation nötigen Berechnungen nur einmal je Frame durchgeführt. Da die Berechnung von *STFT* und Varianzen den Hauptteil der Rechenzeit ausmacht, ist der Einfluss der verschiedenen Merkmale auf die Gesamtrechenzeit bei allen annähernd gleich gering. Um die Diskrimierungsfähigkeit der einzelnen Merkmale zu untersuchen, wurden in Excel Histogramme zu den bei der Verarbeitung eine Menge von mehreren Tausend Frames entstehenden Merkmalsdaten von Musik- und Sprachdaten erstellt. Diese sind in Abbildung 6.1 dargestellt.

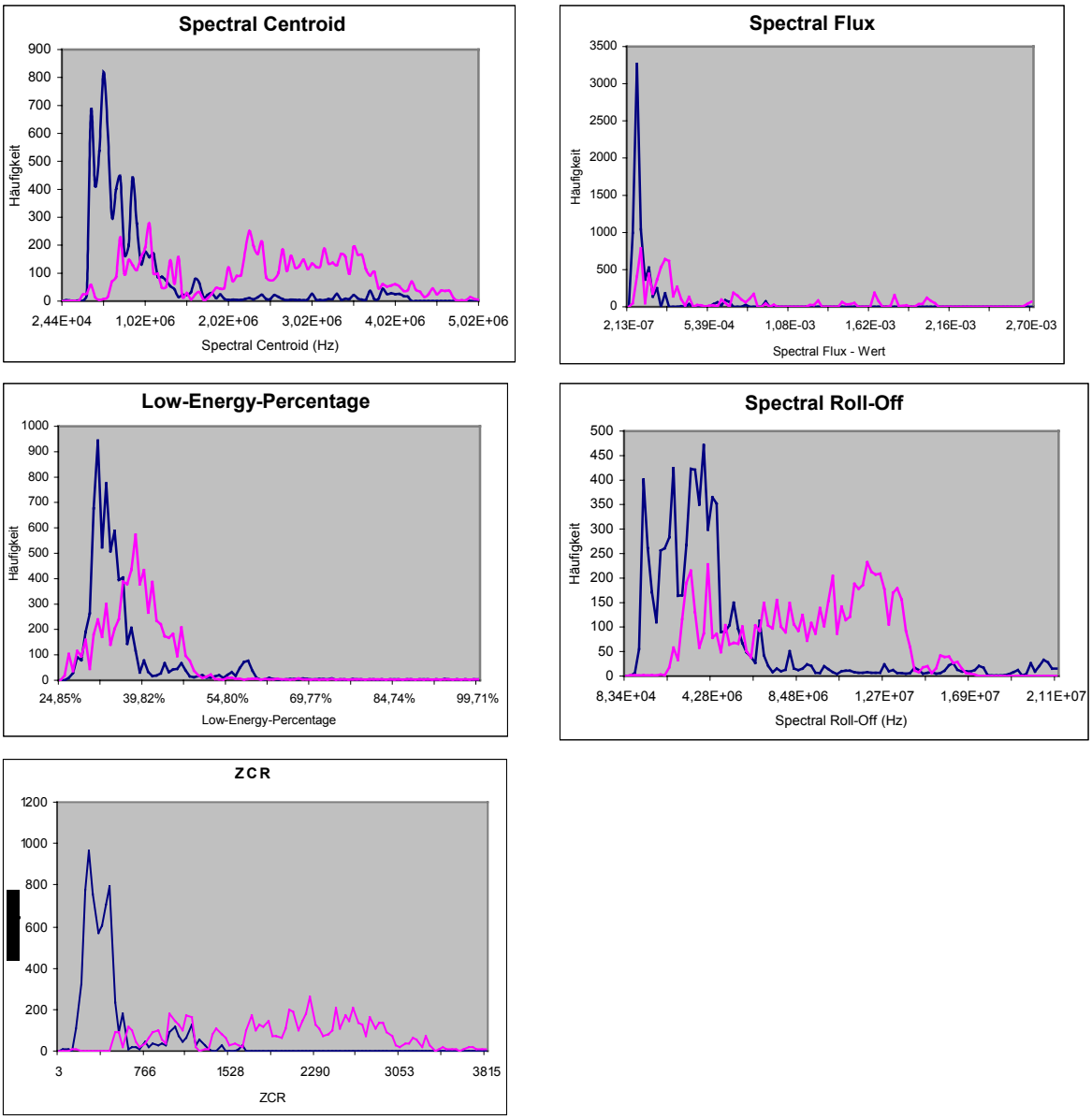
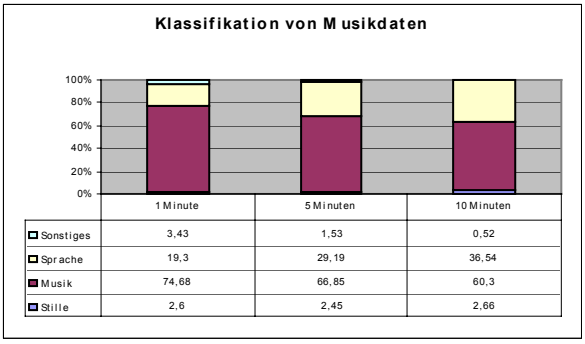
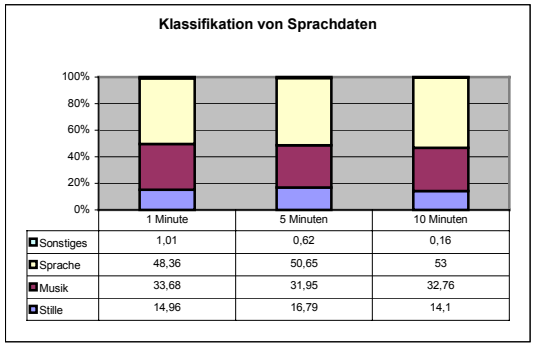


Abb. 6.1: Histogrammdarstellungen zur Diskrimierungsleistung der einzelnen Merkmale. Dabei sind jeweils vertikal die Häufigkeit und horizontal die sortierten Merkmalswerte (in je 100 Bins) aufgetragen. Die Merkmale der Musikdaten sind jeweils blau, die der Sprachdaten rosa markiert.



	1 Minute	5 Minuten	10 Minuten
Sonstiges	3,43	1,53	0,52
Sprache	19,3	29,19	36,54
Musik	74,68	66,85	60,3
Stille	2,6	2,45	2,66



	1 Minute	5 Minuten	10 Minuten
Sonstiges	1,01	0,62	0,16
Sprache	48,36	50,65	53
Musik	33,68	31,95	32,76
Stille	14,96	16,79	14,1

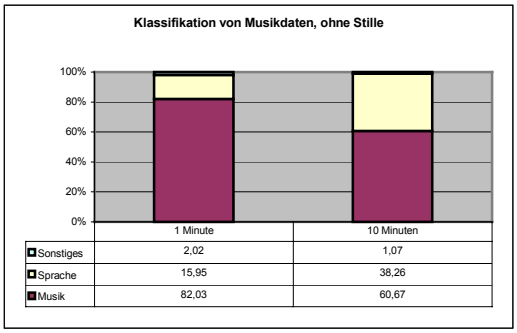
Abb. 6.2: Klassifikationsleistung jeweils als Diagramm und als Matrix. Auf der linken Seite ist die Klassifikation von Musikdaten, auf der rechten die von Sprachdaten abgebildet. Zu sehen sind korrekte und falsche Zuordnungen in Abhängigkeit der Verteilung der Grunddaten auf Trainings- und Testdaten. Die Zeitangaben entsprechenden dem Anteil der zum Training verwendeten Daten der jeweiligen Klasse. Klassifikationsparameter: 44,1kHz Abtastrate, k=5, 3-Frame-Nachbearbeitung (siehe Abschnitt 6.1.6)

In der Abbildung zeigt sich, dass jedes Merkmal mehr oder weniger gut getrennte Wertebereiche bezüglich der beiden Klassen besitzt.

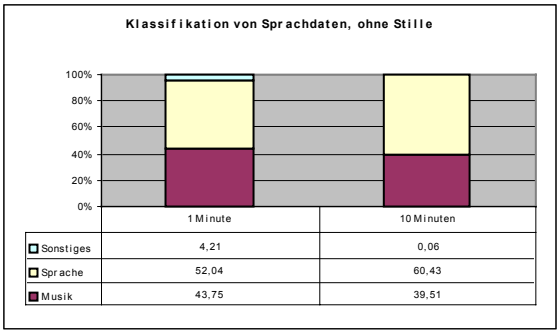
6.1.3 Einfluss der Anzahl von Trainings- und Testdaten

In diesem Test wurde untersucht, in welchem Maß die Anzahl der Trainings- und Testdaten die Klassifikationsleistung beeinflusst. Dazu wurden die 20 Minuten Daten jeder Klasse nacheinander in zwei verschiedene disjunkte Mengen von Trainings- und Testdaten unterteilt und die resultierende Klassifikationsleistung für die beiden Klassen separat gemessen. Alle Varianten wurden mehrfach kreuzvalidiert, um den Einfluss einzelner Audiofragmente zu minimieren. Die Klasse *Sonstiges* enthält die unklassifizierten Merkmalsvektoren. Dieses Verfahren wurde auch in den nachfolgenden Parametertests verwendet. Das Ergebnis ist in Abbildung 6.2 dargestellt. Zu erkennen ist, dass Sprachdaten zwar mit steigender Menge an Trainingsdaten besser erkannt werden, dass Musik jedoch immer weniger korrekt eingeteilt wird. Dabei nimmt die Qualität der Zuordnungen für Musik schneller ab als sie für Sprache zunimmt, vermutlich bedingt durch die Wahl der Trainingsdaten. Auffällig ist weiterhin die hohe *Falschklassifikation* von Sprache als Stille. Dies ist insofern nicht verwunderlich, als Sprache im Allgemeinen viele Lücken im Sprachfluss beinhaltet, die als Stille interpretiert werden können. Um den Einfluss der Erkennung von Stille zu eliminieren, wurde eine vereinfachte zweite Testreihe durchgeführt, die mit der gerade vorgestellten bis auf das ausgeschaltete Stille-Erkennungsmodul identisch ist. Die in Abbildung 6.3 gezeigten Ergebnisse bestätigen den Trend der ersten Messungen.

In den nachfolgenden Tests wird daher, falls nicht anders angegeben, eine Trainingsmenge von je einer Minute pro Klasse verwendet und die Klassifikationsleistung mittels 19 Minuten Testdaten eruiert. Die Erkennung von Stille ist deaktiviert.

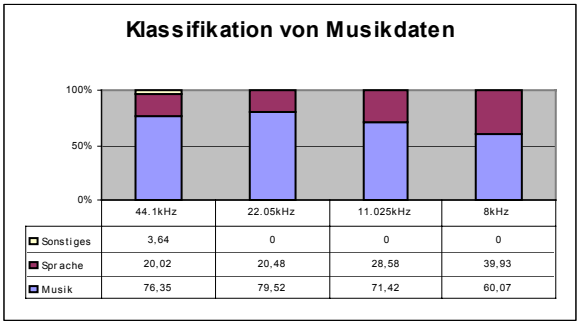


	1 Minute	10 Minuten
Sonstiges	2,02	1,07
Sprache	15,95	38,26
Musik	82,03	60,67



	1 Minute	10 Minuten
Sonstiges	4,21	0,06
Sprache	52,04	60,43
Musik	43,75	39,51

Abb. 6.3: Klassifikationsleistung ohne Stille-Erkennung jeweils als Diagramm und als Matrix. Alle Parameter sind identisch mit den in Abb. 6.2 gezeigten Messungen.



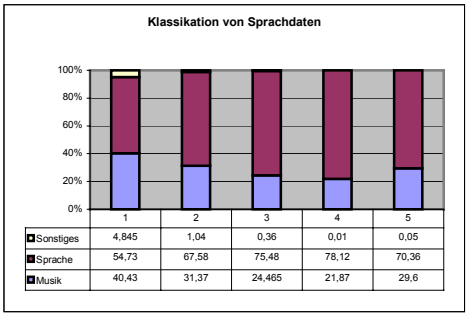


Abb. 6.4: Klassifikationsleistung jeweils als Diagramm und als Matrix. Klassifikationsparameter: k=5, 3-Frame-Nachbearbeitung (siehe Abschnitt 6.1.6)

6.1.4 Einfluss verschiedener Abtastraten

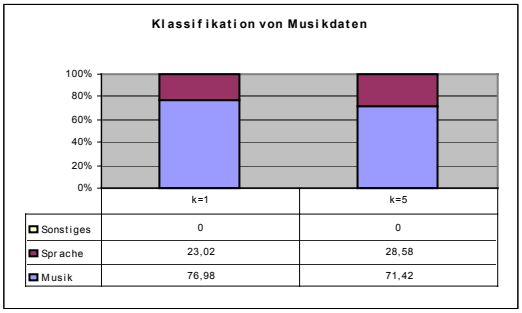
Da Audiodaten in sehr unterschiedlichen Qualitätsstufen zu finden sind, wurde in diesem Test die Klassifikationsleistung bei Veränderung der Abtastrate untersucht, wie in Abbildung 6.4 dargestellt. Dabei wurden sowohl Trainings- als auch Testdaten mit derselben Abtastrate verwendet. Interessanterweise scheint die Klassifikation von Musikdaten am besten bei einer Abtastrate von 22,05 kHz zu funktionieren, während es bei Sprachdaten 8 kHz sind. Als Mittelweg werden daher in den nächsten Tests meist Daten mit einer Abtastrate von 11,025 kHz verwendet.

6.1.5 Einfluss der Fensterweite

Getestet wurden die in den anderen Tests verwendete Fensterweite von 512 Samples, mit 384 Samples Überlappung, sowie die entsprechenden Paare 256/128 und 1024/512. Wie schon von Scheirer & Slaney [93] vermutet, konnte kein nachhaltiger Einfluss der Fensterweite oder der Überlappung auf die Klassifikationsleistung festgestellt werden.

6.1.6 Vergleich verschiedener Nachbearbeitungsmechanismen

In der Publikation von El-Maleh et. al [34] wird ein Verfahren beschrieben, das darauf abzielt, das schnelle Wechseln von Klassenzuordnungen mittels eines laufenden 3-Frame-Votings in der Nachverarbeitung zu unterbinden. In einem Test wurde dieses Verfahren sowohl einem modifizierten Ansatz als auch der Klassifikation ohne entsprechende Nachbearbeitung gegenüberstellt. Das modifizierte Verfahren führt ein Voting auf der Menge von Merkmalsvektoren durch, die in den vorigen 250ms erzeugt wurden. Bei einer Fensterweite von 512 und einer Überlappung von 384 Samples sind dies bei 44,1 kHz ungefähr 28 Vektoren. Die Ergebnisse der drei Verfahren variierten allerdings nur um wenige Zehntel-Prozentpunkte. Im Fall des modifizierten Verfahrens wird jedoch eine bis zu 28-fache Verringerung der Anzahl zu versendender Eventausgaben des Filters erreicht, was der Performanz zu Gute kommen dürfte. Daher verwenden nachfolgende Tests dieses modifizierte Verfahren zur Nachverarbeitung von Klassifikationsereignissen.



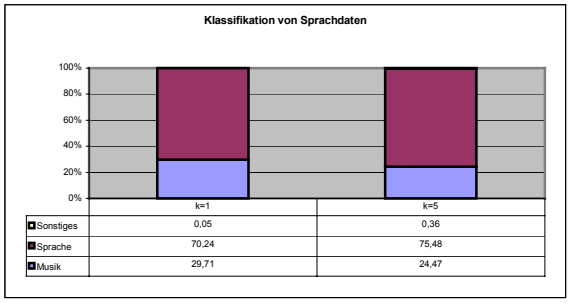


Abb. 6.5: Klassifikationsleistung jeweils als Diagramm und als Matrix.

6.1.7 Einfluss der Parameter des k-NN-Verfahrens

In diesem Test wurde untersucht, wie sich die Klassifikationsleistung des Filters verändert, wenn verschiedene Werte für *k* verwendet werden, also für die Anzahl der im k-NN-Klassifikationsverfahren dem Suchvektor nächstliegenden Trainingsvektoren. Die Ergebnisse sind in Abbildung 6.5 dargestellt und zeigen ein uneinheitliches Bild, in dem sich für verändernde *k*-Werte die Klassifikationsleistung von Musik- und Sprachdaten jeweils entgegengesetzt verbessert bzw. verschlechtert.

6.1.8 Parallelklassifikation mit verschiedenen Abtastraten

In diesem Test wurde untersucht, ob sich mittels einer Parallelklassifikation durch zwei entsprechende *Pipelines* eine Verbesserung der Klassifikationsleistung erreichen lässt. Grundlage dafür war die in Abschnitt 6.1.4 erläuterte unterschiedliche Beobachtung, dass Sprache bei niedrigeren Abtastraten, Musik dagegen bei höheren Abtastraten besser erkannt wird. An dieser Stelle kommt die flexible durch GenMAD ermöglichte Signalverschaltung zum Einsatz, wie in Abbildung 6.6 aufgezeigt. Da zum Zeitpunkt des Testens kein hochwertiges Filter zum Echtzeit-Downsampling der verwendeten 19 Minuten langen Test-Audiodatei vorhanden war, wurde diese extern mit CoolEdit auf 8 kHz heruntergerechnet und als separate Datei eingebunden. Weiterhin kamen in jeder Pipeline Equalizer zum Einsatz, die dem jeweiligen Klassifikator sowohl im Trainings- als auch im Erkennungsmodus vorgeschaltet waren. Ziel der Verschaltung war es, je eine Pipeline zur Erkennung von Musik und Sprache zu verwenden, deren Ausgaben durch ein *MonitoringEventMerger*-Filter zusammengeführt werden, wobei die Signale und Klassifikatoren jeder Pipeline auf höchste Klassifikationsleistung der jeweiligen Klasse optimiert werden. Die Ergebnisse dieses Tests sind in Abbildung 6.7 visualisiert.

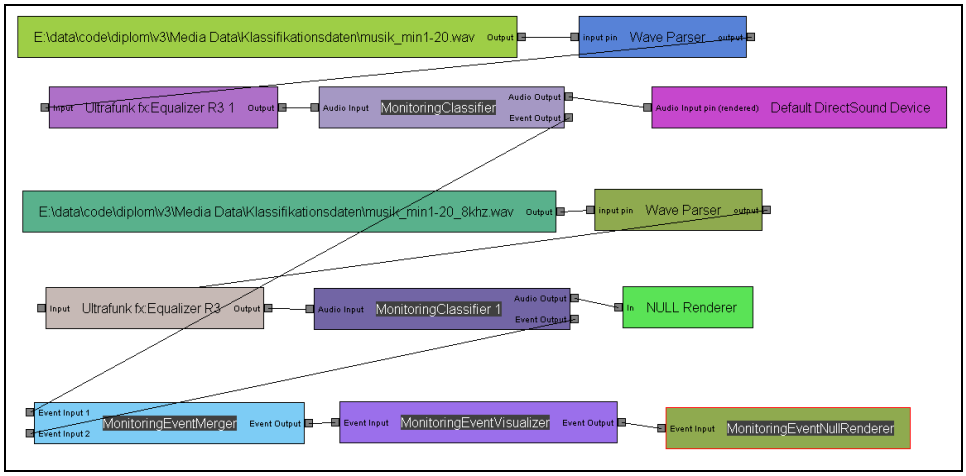


Abb. 6.6: GenMAD-Projekt mit zwei parallel arbeitenden Klassifikationspipelines.

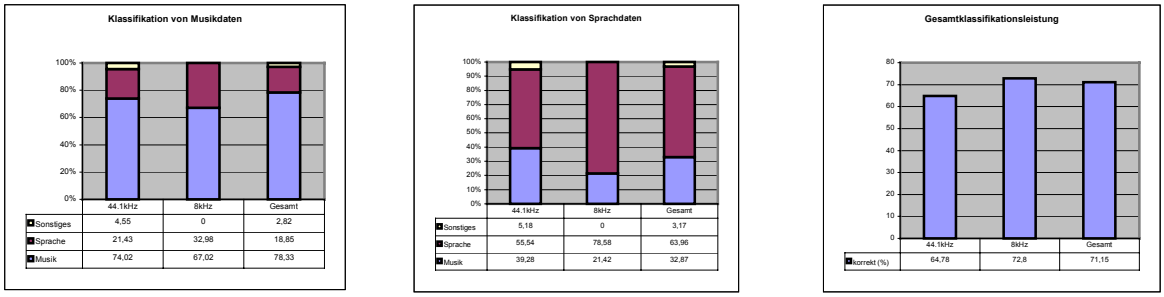


Abb. 6.7: Links und in der Mitte: Klassifikationsleistung jeweils als Diagramm und als Matrix. Rechts: Gesamtleistung. Klassifikationsparameter: k=5, Merger: Diffusion=5, Gewichtung=0, SegmentMindestLänge=0

Wie erwartet erreichten beide Pipelines akzeptable Ergebnisse in ihren Spezialgebieten, die jedoch durch die jeweils entgegengesetzte Klassifikationsaufgabe wieder etwas entwertet wurde. Wie aus dem Gesamtüberblick erkennbar, ist die Leistung des 8 kHz-Klassifikators sogar etwas höher als die der Pipeline-Kombination.

Bemerkenswert ist darüber hinaus, dass das Kombinatorfilter zu einer höheren Gesamtleistung führt als der Mittelwert der beiden Pipelines, so dass durch geeignetes Abstimmen der Merger-Parameter sowie durch optimierteren Einsatz der Equalizer (und weiterer Vorverarbeitungsschritte) eine höhere Leistung zu erwarten sein sollte. Der Vorteil der GenMAD-Plattform ist hier die Möglichkeit, mit wenig Aufwand neue Signalverarbeitungs-kaskaden zu testen. Auch wenn sich in diesem Test noch keine Verbesserung der Leistung ergab, wurde das Prinzip doch im nachfolgenden Abschlusstest eingesetzt.

6.1.9 Klassifikationsleistung anhand von Radiomitschnitten

In diesem abschließenden Test sollte die Gesamtklassifikationsleistung anhand von zwei Radiomitschnitten ausführlich eruiert werden. Zunächst wurde ein 45 Minuten langer Mitschnitt eines per Internet mit 48 kbps gestreamten Radiosenders mit Schwerpunkt Rockmusik (*StarFM 87,9*, Berlin) untersucht, wobei der Projektaufbau dem in 6.1.8 gleicht. Anders als in den vorigen Funktionstest handelt es bei den Testdaten also nicht um Daten von jeweils nur einem Typ, vielmehr wechseln sich Musik und Sprache wie in vom Radio gewohnter Weise ab. Daher wurden der Parameter des Kombinatorfilters auf eine Mindest-Segmentlänge von vier Sekunden angepasst und die Gewichtung der mit 44,1 kHz arbeitenden Pipeline auf 1.5 erhöht, da durch die minderwertige Güte des Eingangssignals mit einer schlechten Klassifikationsleistung der anderen Pipeline zu rechnen war.

Um die Klassifikationen bewerten zu können, wurden sie mit einer manuell durchgeführten Unterteilung verglichen, indem ein Excel-Makro nachträglich korrekte und fehlerhafte Zuordnungen der beiden Pipelines und des Gesamtergebnisses untersuchte. Die dabei erstellten Logdaten erlauben darüber hinaus einen Einblick in die Ursachen fehlerhafter Klassifikationen. Die Ergebnisse sind in Tabelle 6.8 aufgeführt. Die Gesamtleistung wurde anhand der Ausgabe des Kombinatorfilters gemessen. Da keine Klassifikation in *Sonstiges* oder *Stille* stattfand, sind diese Klassen nicht aufgeführt.

	44,1 kHz	**8 kHz**	**Kombination**
Gesamte korrekte Klassifikation	82,38	66,82	83,77
Gesamte falsche Klassifikation	17,62	33,18	16,23
Musik korrekt klassifiziert	88,56	69,84	90,52
Musik falsch klassifiziert	11,44	30,16	9,48
Sprache korrekt klassifiziert	70,11	60,85	69,18
Sprache falsch klassifiziert	29,89	39,15	30,82

Tab.6.8: Übersicht über die Klassifikationsleistung (in %) der beiden Pipelines und der Kombination im Radiotest 1.

	44,1 kHz	8 kHz	Kombination
Gesamte korrekte Klassifikation	56,6	75,85	63,71
Gesamte falsche Klassifikation	43,4	24,15	36,29
Musik korrekt klassifiziert	49,56	74,51	57,23
Musik falsch klassifiziert	50,44	25,49	42,77
Sprache korrekt klassifiziert	82,47	80,79	84,92
Sprache falsch klassifiziert	17,53	19,21	15,08
Stille korrekt klassifiziert	66,67	77,78	86,67
Stille falsch klassifiziert	33,33	22,22	13,33

Tab.6.9: Übersicht über die Klassifikationsleistung (in %) der beiden Pipelines und der Kombination im Radiotest 2.

Auffällig ist hier zum einen, dass der auf 44,1 kHz operierende Klassifikator wie vermutet in beiden Fällen signifikant besser arbeitet als der 8 kHz-Klassifikator. Dies scheint auf die ohnehin hohe Komprimierung der Testdaten zurückzuführen zu sein, so dass die Verbindung von dem mit 48 kHz kodierten Signal und dem Downsampling auf 8 kHz eine zu starke Verzerrung erzeugte. Zum anderen wird hier im Gegensatz zum Test in 6.1.8 eine Verbesserung der Gesamtleistung in Folge der Kombination der beiden Pipelines sichtbar.

In den Logdaten zeigt sich, dass alle drei gemessenen Ereignisreihen aus langen Passagen mit einer Klassifikation bestehen, die in mehr oder weniger häufigem Maß durch *Ausreißer* (Fehlklassifikationen) von ca. 1-20 Sekunden Länge unterbrochen werden. In der 44,1 kHz-Pipeline sind die Ausreißer seltener und von gleichmäßigerer Art und Länge als in der mit 8 kHz arbeitenden Pipeline, wo oftmals ein Wechsel zwischen 2-3 Sekunden langen Folgen zu beobachten ist. Die Häufigkeit dieser Ausreißer scheint das Hauptkriterium der Klassifikationsleistung zu sein, da komplette Falschklassifkationen über lange Zeiträume selten sind. Parametrisiert durch den Diffusionswert und die Segmentmindestlänge glättet das MonitoringEventMerger-Filter diese Ausreißer bis zu einem gewissen Maß. Hier kann durch eine feinere Anpassung des Parameter des Kombinatorfilters sicherlich die Leistung weiter erhöht werden.

Weiterhin sichtbar wird, dass die 8 kHz-Pipeline oftmals eine Verzögerung gegenüber der manuell durchgeführten Klassifikation ausweist, wenn dort ein Klassenwechsel stattfindet.

In einem zweiten Test wurde eine in CD-Qualität aufgenommene Radiosendung von ebenfalls 45 Minuten Länge dem gleichen Testverfahren unterzogen. Dabei wurde bewusst ein Sender (*JamFM*) gewählt, der wie viele der großen Radiostationen fast ausschließlich R'n'B und souligen Hiphop spielt. Die Teilergebnisse sind in Tabelle 6.9 zusammengefasst. Auch hier wurden keine Daten als *Sonstiges* klassifiziert. Da der Anteil falsch klassifizierter Stille-Daten minimal ist, wird auf eine Konfusionsmatrix zugunsten der Vergleichbarkeit verzichtet.

Die sowohl schlechtere Klassifikationsleistung insgesamt als auch das im Gegensatz zur 8 kHz-Pipeline schlechtere Ergebnis der Kombination erklären sich aus dem Ablauf-Log: Anders als im Rock-Test findet man hier nicht nur ein häufiges Springen zwischen den Klassen, sondern auch lange falsch klassifizierte Passagen. Insbesondere wurde, wie auch aus Tab.6.9 zu entnehmen, oft Musik als Sprache eingeordnet. Stille dagegen wurde meist korrekt erkannt, in manchen Fällen jedoch mit einer leichten Verzögerung, die allerdings auch in den anderen beiden Klassen mit bis zu vier Sekunden auftrat.

Die Falschklassifikation von Musik als Sprache scheint bei Analyse des Audiomaterials durch die Art der Musik begründet: So ist es in den aktuellen Fassungen der genannten Genres weit verbreitetes Stilmittel, den Instrumentalanteil auf sehr minimale percussive Elemente und punktuelle Basseinsätze zu reduzieren und die Stimmen stark hervorzuheben. Durch das Fehlen konstanter Hintergrundmusik wurde daher scheinbar der Stimmanteil oft als Sprache klassifiziert.

Da auf der anderen Seite Radioansagen und sogar Nachrichtensendungen mehr und mehr mit Musik unterlegt werden, schwindet die auch für Menschen leicht zu trennende Grenze zwischen Musik und Sprache. Abhilfe könnte möglicherweise der geschickte Einsatz von Rhythmikanalysemodulen und -merkmalen bringen, wie das von Tzanetakis et al. zur musikalischen Genreklassifikation eingesetzte Verfahren [108]. Auch die von Scheirer et al. [92], Alonso et. al [3] und anderen Arbeitsgruppen veröffentlichten Publikationen beschäftigen sich mit dem Gebiet der Tempo- und Beatberechnung bzw. -schätzung, so dass Fortschritte bezüglich dieser Klassifikationsproblematik zu erwarten sind.

6.2 Performanz des GenMAD-Systems

In diesem Abschnitt sollen die Ergebnisse verschiedener Tests präsentiert werden, anhand der die Performanz der entwickelten Filter und der GenMAD-Applikation überprüft wurde. Das Testsystem bestand aus einem mit 2,66GHz getakteten Pentium-4-Prozessor und einer Hauptspeichergröße von 512MB. Die abgebildeten Prozessorlastangaben sind gemittelte Schätzwerte.

6.2.1 Performanz der entwickelten Filter

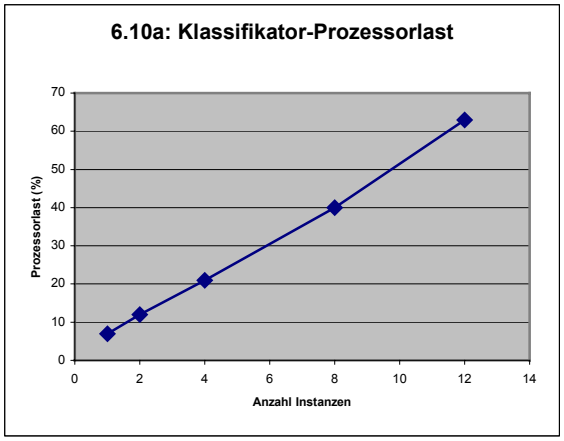
Das einzige der entwickelten Filter, dessen Speicherbedarf nicht nahe bei Null liegt, ist das MonitoringClassifier-Filter: Jede Instanz des MonitoringClassifier-Filters belegt einen Hauptspeicherplatz von etwa 60MB. Dieser Bedarf entsteht durch Variablenfelder konstanter Größe, die für die Verwaltung der Baumdaten benötigt werden.

Ein ähnlich einseitiges Bild zeigt sich auch beim Test der Prozessorlast: Selbst bei Verschaltung von zwölf Filter-Instanzen in einem GenMAD-Projekt benötigt keines der rein ereignisbasierten Filter mehr als ein oder zwei Tausendstel der CPU-Leistung. Auch in Kombination, also z.B. in Form eines Projektes von je zwölf MonitoringXMLReadern, MonitoringEventSplittern und MonitoringXMLWritern, beträgt die Gesamtlast etwa 1%.

Anders im Fall des Klassifikatorfilters: Falls das gesamte Projekt über nur eine Mediendatei in CD-Qualität durch ein nachgeschaltetes Audiosplitterfilter mit Daten beliefert wird, ist die Prozessorlast wie zu erwarten eine lineare Funktion der Zahl von Instanzen, wie in Abbildung 6.10a dargestellt: Jede Instanz benötigt etwa 5% Prozessorleistung. Verändert man die Abtastrate der Eingabedatei, so zeigt sich wiederum ein lineares Verhältnis, wie in Abbildung 6.10b dargestellt. Die dort abgebildeten Werte entsprechen der Last in einem Projekt mit zwölf Klassifikatorfiltern. Wird jedes Filter mit einer eigenen Audiodatei gespeist, so sind die Zusammenhänge nahezu identisch mit den in 6.10b abgebildeten Daten. Es scheint also – einen schnellen Dateizugriff vorausgesetzt – keinen Unterschied zu machen, ob aus verschiedenen Datei gestreamt wird oder ob ein Audiosplitterfilter einen eingehenden Datenstrom in verschiedene Datenströme aufteilt.

Auch der Einfluss der Bitzahl je Sample eines Datenstroms auf die Prozessorlast scheint einer linearen Funktion zu folgen. Im Fall von vier Klassifikatorfiltern liegt die CPU-Auslastung bei 16 Bit Auflösung bei 19%, während es bei 8 Bit nur 11% sind. Der Einfluss mehrerer Datenkanäle ist zu vernachlässigen, da nur der linke Monokanal verwendet wird.

Um auch den Einfluss des Audentify!-Filters zu testen, wurden in gleicher Weise wie im Falle des Klassifikatorfilters mehrere Instanzen in ein Projekt eingefügt. Die Filter greifen dabei auf verschiedene Indexdateien gleichen Inhalts zu. Aufgrund der großen, durch die DLL angeforderten Pufferlänge zeigt sich ein konstantes Bild von ca. 2% Last zwischen den Bearbeitungsphasen der DLL. Wird der Pufferinhalt bearbeitet, kommt es allerdings zu Leistungsspitzen von bis zu ca. 18% CPU-Last. Ab einer Zahl von drei Instanzen stürzt die DLL jedoch reproduzierbar ab, wodurch weitere Tests verhindert werden.



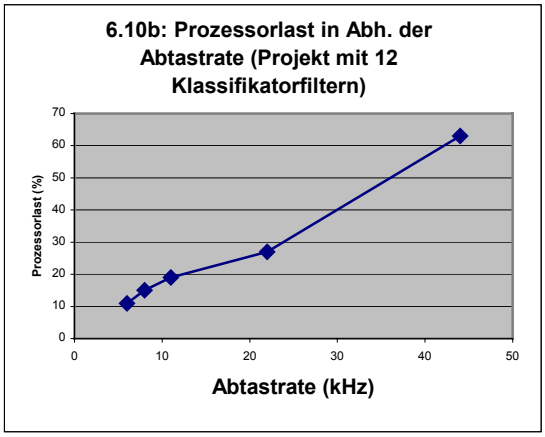


Abb. 6.10a, b

Die Prozessorlast von Filter-Pipelines berechnet sich durch die modulare Filterstruktur von GenMAD direkt aus der den oben untersuchten Eigenschaften des Datenstromtyps und der Last der enthaltenen einzelnen Filter.

Die entwickelten Filter besitzen also insgesamt gute Skalierungseigenschaften. Durch etwa den Einsatz von Pipelinekombinationen, wie in Kapitel 6.1 dargestellt, lassen sich dabei die Vorteile der unterschiedlichen Klassifikationsleistungen und der skalierbaren Prozessorlast vereinen. Die im Vergleich zu Audiodaten sehr geringe Datengröße von Ereignisdaten erlaubt darüber hinaus den zahlreichen, flexiblen Einsatz ereignisbasierter Filter, mittels denen ein Verarbeitungsnetzwerk zur Unterstützung von Klassifikation oder Identifikation umgesetzt werden kann. Die ungenutzte Prozessorleistung kann dann effektiv in Form hochwertiger Vorverarbeitungsfilter eingesetzt werden, wie in Kapitel 6.3 gezeigt.

6.2.2 Performanz von GenMAD

Die Laufzeitleistung von GenMAD wird fast ausschließlich durch die im Datenflussgraphen enthaltenen Filter bestimmt. Lediglich die Visualisierung der verarbeiteten Daten benötigt in wesentlichem Maße Rechenzeit. Auch ohne eingehende Daten schlägt das offene Visualisierungsfenster mit ca. 4% Prozessorlast zu Buche. Für die datenbezogene CPU-Last muss zwischen Ereignis- und Audiodaten unterschieden werden: Die Audiodatenanzeige benötigt unabhängig vom Rest des Projektes ca. 10% der Prozessorleistung. Ist der Audiofokus keinem Filter zugewiesen, wird keine Prozessorlast verursacht. Die Darstellung von Eventdaten scheint ebenfalls einem linearen Verlauf zu folgen, bei dem je Event-Visualisierungsfilter 0,5% der Prozessorleistung benötigt werden.

6.3 Anwendungen

GenMAD bietet vielfältige Ansätze für Monitoring-Anwendungen. Diese ergeben sich vor allem aus den Mitteln zur flexiblen Erzeugung und Bearbeitung des Datenflussgraphen in Kombination mit einer großen Menge von hochwertigen, durch Dritte entwickelten Filtern. Die im Monitoring stets präsenten Bereiche Vorverarbeitung, Analyse und Nachverarbeitung profitieren dabei alle von den gebotenen Möglichkeiten:

Die Unterstützung aller gängigen Audioformate und die Verwendbarkeit von Capturingfilter erlauben den Einsatz in vielen verschiedenen Umgebungen. Ein Einsatzgebiet besteht z.B. in der Verwendung eines in Echtzeit eingehenden Videosignals, dessen Audiokanal für Monitoringzwecke eingesetzt wird. Hinzu kommt die Möglichkeit, Dateien über das Internet

miteinzubeziehen. Auf diese Weise lassen sich auch servergestützte Projekte erzeugen, um z.B. öffentlich publizierte Nachrichtensendungen oder Onlineradiosendungen zu verwenden.

In der Vorverarbeitung können beliebige Kombinationen hochwertiger Filterungs- und Effektmodule eingesetzt werden, um eingehende Datenströme zu transformieren. Ein Beispiel dafür ist etwa der in Abbildung 6.6 dargestellte Einsatz eines Equalizerfilters, der gezielt bestimmte Frequenzbereiche verstärkt bzw. vermindert, um den nachfolgenden Klassifikationsfiltern ein möglichst gut auf ihren Zweck zugeschnittenes Frequenzspektrum zu bieten. In ähnlicher Art können Sprachverarbeitungsfilter wie De-Esser zum Einsatz kommen, die Störlaute reduzieren. Ein übliches Problem bei der Verwendung von in Echtzeit über Capturingfilter eingehenden Signalen ist Rauschen, welches durch De-Noiser-Module vermindert werden kann. In vielen publizierten Verfahren werden Hoch- oder Tiefpassfilter eingesetzt, um das Frequenzspektrum einzugrenzen.

Auf der anderen Seite ist es durch künstliches Einstreuen von Störungen und Signalverzerrungen genauso möglich, eine Simulationsumgebung zu erzeugen, die Parameterabstimmungen im Voraus für bestimmte Einsatzzwecke ermöglicht, etwa um die in einer Live-Aufnahme oft auftretenden Hintergrundgeräusche zu vermindern. Durch den Einsatz von Hall- und Verzögerungseffekten, vielleicht sogar in Verbindung mit zusätzlichen Beispieldaten typischer Hintergrundgeräusche, läßt sich ein fertiges Projekt für den Live-Einsatz vorbereiten, das den eventuellen Gegebenheiten Rechnung trägt.

Ein großes Potential bietet auch die Verwendung von Pipelines, die durch Splitterfilter verteilte Datenströme unabhängig von anderen Signalwegen bearbeiten können. Dies kann z.B. zu Vorverarbeitungszwecken im Sprachmonitoring eingesetzt werden, indem eine Pipeline durch ein Hochpassfilter nur die oberen Frequenzbereiche eines Signals manipuliert, während in einer zweiten Pipeline unter Verwendung einer Tiefpassfilters der Bassanteil bearbeitet wird.

Verwandt mit diesem Ansatz sind Projekte zur Analyse räumlicher Informationen. Dazu können Signale in den beiden Kanälen eines Stereosignals separat analysiert werden, um eine Signalquelle zu lokalisieren, etwa ein Vogelruf in einer Monitoringaufnahme in der Natur.

Neben der Parallelverarbeitung reiner Audiodaten lassen sich auch Kombinationen aus Audio- und Eventdaten nutzen, um einen Mehrwert aus der Verwendung von Pipelines zu ziehen. Ein wichtiges solches Beispiel ist der in Abbildung 6.6 dargestellte und in den Anschnitten 6.1.8 und 6.1.9 untersuchte Ansatz, ein Audiosignal auf verschiedene Weisen vorzuverarbeiten, separat zu klassifizieren und die Eventdaten mittels eines Kombinatorfilters wieder zusammenzuführen. Auf diese Weise lassen sich Analyse- und Verarbeitungsmodule optimal aufeinander abstimmen und können, wie in 6.1.9 gezeigt wird, zu höheren Klassifikationsleistungen führen.

In ähnlicher Weise ist der Aufbau einer parallel arbeitenden Filterbank möglich, die Analysen verschiedener Frequenzbänder zusammenführt, wie in Abbildung 6.11 dargestellt. Dazu wird das Signal zunächst auf vier Pipelines aufgeteilt, von denen jede das Signal durch ein Filtermodul auf einen bestimmten Frequenzbereich eingrenzt, das veränderte Signal klassifiziert und die generierten Eventdaten durch eine Kaskade von Kombinatorfiltern an einen XMLWriter-Filter übergibt. Durch das Hinzufügen zusätzlicher EventExcluder- oder Includerfiltern läßt sich das Auftreten bestimmter Ereignisse in einem Frequenzband herausfiltern.

Durch die Möglichkeit Eventdaten aufzuzeichnen, lassen sich z.B. Identifikationsdaten archivieren und später zu Vergleichszwecken einsetzen, etwa im Rahmen einer Szenenanalyse. Ein solches Projekt ist in Abbildung 6.12 dargestellt. Dort werden zwei Pipelines zusammengeführt: In der einen werden in Echtzeit in den Rechner eingehende Signale per Audentify! auf bekannte Audiostücke hin untersucht, die daraus entstehenden Eventdaten mittels eines EventIncluder-Filters auf eine Menge erlaubter Ereignisse eingegrenzt, zur Visualisierung an GenMAD und anschließend an den Kombinator übergeben. Parallel dazu werden archivierte Eventdaten eingelesen, visualisiert und ebenfalls in den Kombinator eingespeist, der das Ergebnis

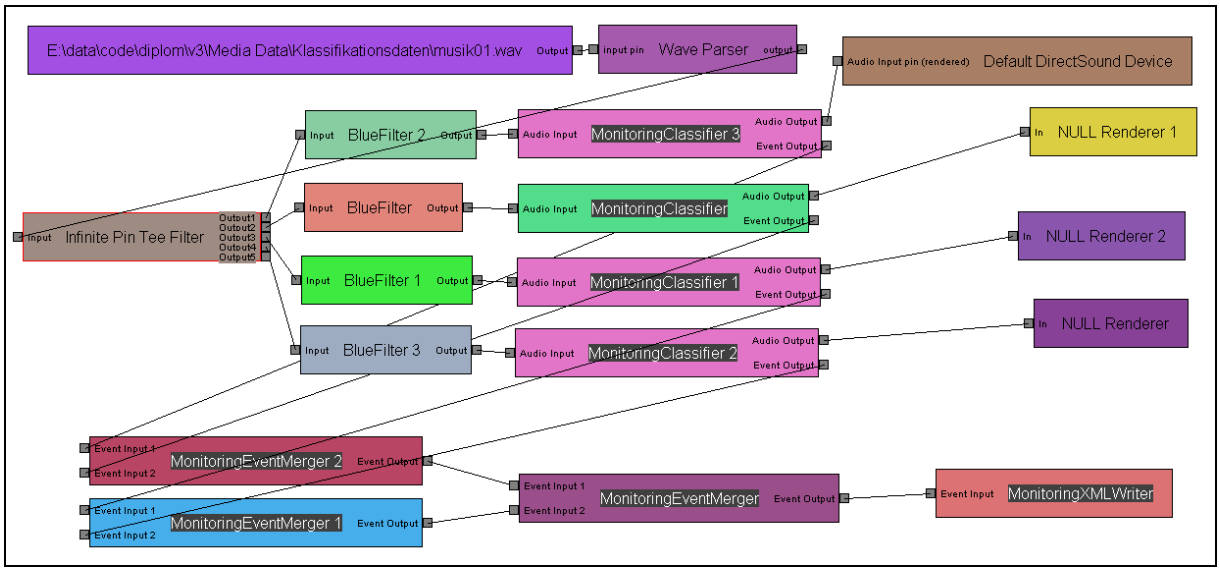


Abb. 6.11: Klassifikation von Frequenzbändern

der Kombination als dritten Visualisierungstrom an GenMAD übergibt. Auf diese Weise können im Visualisierungsfenster von GenMAD die drei Visualisierungssignale in Echtzeit verglichen werden.

Auch die Nachbearbeitung profitiert von den genannten Möglichkeiten: Ebenso wie Eventdaten können auch Audiodatenströme archiviert werden, um sie später wiederzuverwenden oder in anderen Applikationen einzusetzen. Durch die bereits angesprochenen EventEx- und Includerfilter können Ereignisse manipuliert, durch EventSplitter und -Merger zerteilt und zusammengeführt werden.

Wie in Abbildung 6.12 gezeigt, bietet der punktuelle Einsatz von EventVisualizerfiltern die Möglichkeit, das Monitoring gezielt auf bestimmte Ereignisse einzugrenzen: Nur gewünschte Ereignisdatenströme und auf Wunsch nur bestimmte darin enthaltene Ereignisse werden angezeigt. Über dieses Verfahren kann z.B. nach dem Auftreten eines bestimmten Werbejingles in einem Onlineradiostream gehorcht werden, ohne durch andere Identifikationsmeldungen den Überblick zu verlieren.
Hier setzt ein weiterer Vorteil an, der im nachfolgenden Kapitel 7 weiter diskutiert wird: Die modulare Erweiterbarkeit von GenMAD durch eigene (mittels des SDK erstellte) Filter. Im genannten Beispielszenario ließe sich durch ein eventbasiertes Filter z.B. ein Emailversand beim Eintreffen des gesuchten Jingles realisieren.

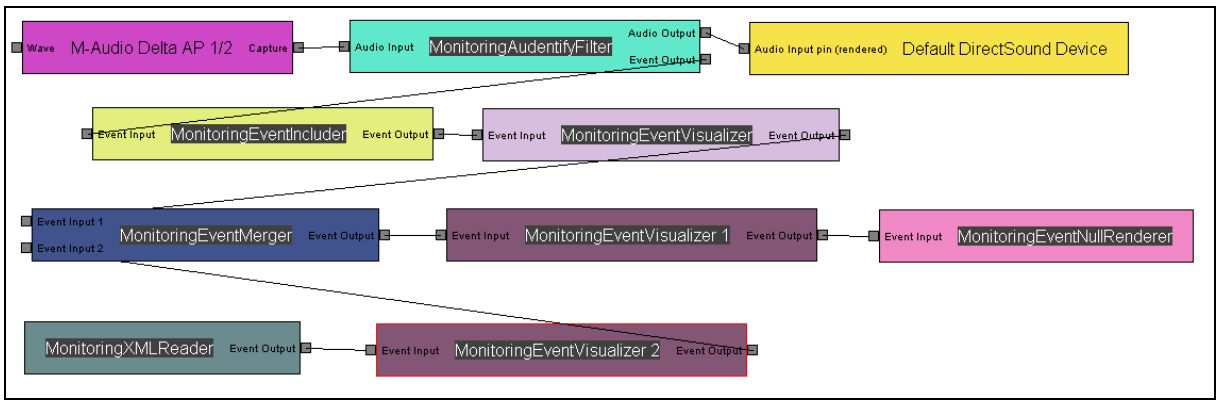


Abb. 6.12: Vergleich von archivierten und Echtzeit-Eventdaten

Kapitel 7

Zusammenfassung, Diskussion und Ausblick

In dieser Arbeit wurde ein generisches Monitoringsystem für akustische Datenströme konzipiert und realisiert. Dieses GenMAD genannte System ermöglicht die flexible freie Verschaltung, grafische Konfiguration und Echtzeitvisualisierung von Signalverarbeitungsmodulen, die auf dem DirectShow-Plugin-Standard beruhen und die optional eine eigens entwickelte erweiterbare Schnittstelle für zusätzliche Funktionalität implementieren können. Unterstützt werden dabei explizit die Verarbeitung von sowohl Audio- als auch Ereignisdaten, die etwa von Klassifikatoren generiert werden können, sowie Möglichkeiten zur Einbindung von Live- und Offlinedaten, wobei letztere lokal und im Internet angesprochen werden können. Darüber hinaus bietet das System die Möglichkeit, Datenströme beider Typen geeignet formatiert in Dateien abzuspeichern.

Neben der zentralen Host-Applikation wurden eine Reihe von Plugins des genannten Typs realisiert, die Datenströme vereinen, verteilen, verändern, erzeugen oder zerstören können. Insbesondere wurde ein Modul zur Klassifikation von Audiosignalen in Stille, Sprache, Musik und Sonstiges erstellt sowie ein von der Arbeitsgruppe Multimedia-Signalverarbeitung entwickelter Algorithmus zur inhaltbasierten Audioidentifikation in ein Plugin eingebunden. Darüber hinaus wurde auch ein Modul realisiert, das eine lineare Segmentierung eingehender Ereignisdatenströme mehrerer Quellen zur Vermeidung widersprüchlicher Daten ermöglicht. Bei allen realisierten Modulen kam ein im Rahmen dieser Diplomarbeit entwickeltes SDK (Software Development Kit) zum Einsatz, welches die Umsetzung von Plugins für das entworfene Monitoringsystem deutlich erleichtern soll. Weiterhin bietet dieses SDK die Möglichkeit, das entworfene Pluginmodell effektiv zu erweitern. Sowohl bei der Erstellung des zugrunde liegenden Konzeptes als auch während dessen Umsetzung wurde mit größter Sorgfalt darauf geachtet, die zu Beginn des Kapitels 4 genannten Anforderungen zu erfüllen.

In der vorliegenden Arbeit wurden in Kapitel 2 Konzepte und bestehende Ansätze zu den Gebieten der inhaltsbasierten Audioidentifikation, der Audioklassifikation und des Audiomonitorings, sowie Anwendungen aus diesen Bereichen vorgestellt. Dabei wurde Wert darauf gelegt, sowohl formale Aspekte als auch umgesetzte Systeme zu präsentieren, um ein umfangreiches Verständnis für das entworfene Monitoringsystem zu ermöglichen. Das für die technische Umsetzung dieses Systems grundlegende Konzept von Multimedia-Frameworks wurde im nachfolgenden Kapitel 3 erläutert und in Form existierender Frameworks konkretisiert, deren Eigenschaften und Funktionsweisen verdeutlicht wurden.
Das grundlegende Konzept des Monitoringsystems wurde in Kapitel 4 formal beschrieben. Weiterhin wurden Unterschiede und Gemeinsamkeiten mit bestehenden Systemen aufgeführt, sowie mögliche konzeptionelle Alternativen diskutiert. Auch zentrale Mechansimen des Systems

und das Zusammenspiel von Host-Applikation und Verarbeitungsmodulen wurden in diesem Kapitel vorgestellt. Ein Überblick über die konkrete Realisierung des Systementwurfs und die dabei umgesetzten Lösungen zentraler Probleme bestimmen den Inhalt von Kapitel 5. Verschiedene Leistungsaspekte und die Laufzeiteigenschaften aller entwickelten Elemente dieses Monitoringsystems wurden anschließend in Kapitel 6 beschrieben. Abschließend wurden verschiedene Anwendungen des entwickelten generischen Monitoringsystems aufgeführt und mögliche weitere Einsatzgebiete und konkrete Lösungen realer Probleme diskutiert.

Der Anhang umfasst sowohl Hinweise zur Weiterentwicklung des Systems und der Module als auch das Benutzerhandbuch und die technische Dokumentation des entwickelten generischen Monitoringsystems.

Wie in den Kapiteln 2.7 und 4.1 dargestellt wird, existiert nach Wissen des Autors zur Zeit kein Monitoringsystem, das die gerade genannten Eigenschaften besitzt. Insbesondere die Möglichkeit, sowohl auf die sehr große Anzahl existierender DirectShow-Signalverarbeitungsmodule zurückzugreifen als auch mit Hilfe des SDK mit geringem Aufwand eigene Plugins zu entwickeln, schafft Raum für neuartige Monitoringanwendungen und erlaubt ein flexibles und produktives Arbeiten ohne aufwendige Einarbeitung. Hinzu kommt die durch das SDK und die grundlegenden Schnittstellen gegebene Erweiterbarkeit des Systems, so dass auch zukünftige Entwicklungen auf diesem Gebiet mit einfließen können. Gleiches gilt natürlich auch für die GenMAD-Applikation, die z.B. leicht um weitere Visualisierungsoptionen ergänzt werden könnte, wie in Abschnitt 7.1 beschrieben wird.

Ein wesentliches Maß für die Praxistauglichkeit eines Systems stellen seine Performanz und Skalierbarkeit dar. Gegenüber einer proprietären Monitoringlösung ist in einem generischen System stets mit einem Overhead zu rechnen, der darauf beruht, dass das System eine große Verschiedenheit von Modulen verarbeiten können muss. Wie in Kapitel 6.2 aufgezeigt, zeichnet sich die GenMAD-Applikation jedoch durch eine äußerst geringe Grundprozessorlast aus, und auch die rein auf Eventdaten arbeitenden Filter benötigen eine lediglich minimale Prozessorleistung. Auch der Einsatz der Klassifikations- und Identifikationsfilter in komplexen Projekten ist problemlos möglich, wobei sich die Klassifikationsfilter zusätzlich durch die Wahl des Audioformates gut skalieren lassen. Trotzdem zeigte das System auch in komplexen Testprojekten eine solide Robustheit. Es sei allerdings angemerkt, dass fremdentwickelte Filter im Fall interner Fehler den Datenflussgraphen so beeinträchtigen können, dass ein Fortsetzen des Datentransports nicht möglich ist. Da DirectShow-Filter COM-basiert sind und durch das entsprechende Windows-Subsystem kontrolliert werden, bestehen prinzipbedingt nur eingeschränkte Möglichkeiten zur Behandlung solch grundlegender Fehler. Auf der anderen Seite – und im Normalfall – führt das COM-Framework aber auch zu einer sichereren Ausführung, da das gesamte grundsätzliche Filtermanagement, das z.B. die Verwaltung von Filtern und die Zuteilung von Ressourcen umfasst, extern durchgeführt wird. Dadurch wird GenMAD ein Arbeiten auf abstrakterer Ebene ermöglicht.

Natürlich stellt die damit thematisch verbundene Festlegung auf DirectX und die Windows-Plattform eine Beschränkung dar. Wie in Kapitel 3.3 dargelegt, existieren jedoch mit Helix, Quicktime und JMF nur drei Multimedia-Frameworks, denen man eine plattformübergreifende Unterstützung zugestehen kann. Keines dieser Frameworks kann jedoch sowohl einen so hohen Verbreitungsgrad, eine vergleichbare Performanz und gleichzeitig eine äquivalent große Menge bereits existierender Verarbeitungsmodule wie DirectX aufweisen. Daher stehen dieser Einschränkung große Vorteile gegenüber, aus denen GenMAD seine genannten Stärken zieht. Die Umsetzung des Systems mit seinen jetzigen Eigenschaften wäre mit keinem anderen Multimedia-Framework möglich gewesen.

Ausblick

Die gesamte GenMAD-Architektur wurde bewusst auf Erweiterbarkeit hin ausgelegt. Entsprechend vielfältig sind die Möglichkeiten, das System durch weitere Entwicklungen zu ergänzen:

Die in der GenMAD-Applikation bereits vorhanden Möglichkeiten zur Visualisierung von Audio- und Eventdaten ließen sich durch ein konfigurierbares System ausbauen, so dass sich der Benutzer die Bestandteile des Visualisierungsfensters flexibel aus einer Menge von Visualisierungselementen zusammenstellen kann. Solche Elemente können z.B. eine Echtzeit-Visualisierung von Audiomerkmalen wie RMS oder ZCR ermöglichen oder per STFT das Frequenzbild eines Audiofilters darstellen. Die dazu verwendeten Algorithmen können entweder direkt implementiert oder in Form von DLLs mit den entsprechenden Filtern geteilt werden. Eine dritte Möglichkeit besteht darin, Filter die Merkmalsvektorfolge direkt an GenMAD versenden zu lassen, wodurch eine Mehrfachberechnung umgangen werden kann. Solche Merkmalsdaten lassen sich z.B. wie in Abbildung 6.1 als Histogramm anzeigen, evtl. sogar mit einem chronologischen Verlauf. Dieser kann durch einen Farbverlauf verdeutlicht werden, bei dem aktuelle Daten mit hoher Farbintensität, ältere Daten dagegen zunehmend blasser dargestellt werden. Noch etwas weiter gedacht läßt sich auch die Darstellung beliebiger filterspezifischer (Echtzeit-)Informationen realisieren, wie etwa in einem Histogramm, das Zeitverschiebungen (*Shift*-Werte) des Audentify-Algorithmus' anzeigt.

Auch das Arbeiten mit Filtern und deren Verschaltung kann – gerade im Hinblick auf Filterpipelines – dadurch vereinfacht werden, dass Filtermodule in Multifilter-Containern zusammengefasst und wie Presets separat abgespeichert und geladen werden können.

Ein weiterer interessanter Punkt ergibt sich aus der Automation von Abläufen: So bietet DirectX eine solche Automation an, mittels der sich Parameterwerte dynamisch ändern lassen. Diese Änderungen lassen sich ebenfalls laden und speichern und bieten so eine Möglichkeit für die Einführung von *Sessions*, bei denen ein bestehendes (statisches) Projekt aus Filtern mit zeitbasierten Dynamikinformationen angereichert wird.

Damit verwandt ist das Konzept der *Stapelverarbeitung*, etwa zur sequentiellen Verarbeitung einer ausgewählten Reihe von Eingabedateien, gesteuert durch ein vom Benutzer erstelltes Skript.

Auch Filter bieten Möglichkeiten für Weiterentwicklungen: So läßt sich durch Bereitstellen entsprechender Funktionen die vollständige Pflege der Audentify!-Parameter über das in GenMAD enthaltene dialogbasierte Konfigurationssystem abwickeln, wodurch Tests vereinfacht werden. Die zur Zeit implementierten Filter-Konfigurationselemente können zu diesem Zweck um Listen, Auswahlboxen und spezifische Buttons erweitert werden.

Um die Flexibilität zu erhöhen und das Prinzip des Monitorings in einen größeren Zusammenhang zu stellen, können z.B. ereignisbasierte Filter entwickelt werden, die bei Eintreffen bestimmter Ereignisse externe Funktionen ausführen, wie etwa zum Emailversand oder zur Generierung akustischer Signale. Bereits bestehende Konzepte lassen sich darüber hinaus erweitern, indem statischen, parameterbasierten Entscheidungsprozessen Möglichkeiten zur Adaption an die eingehenden Daten gegeben werden, z.B. im Fall des MonitoringEventMergers, der so auf schnelles Springen zwischen mehreren Klassen reagieren kann.

Ein weiteres wichtiges Feld bietet der Datenaustausch mit anderen Applikationen, ob lokal oder netzwerkbasiert. Hier ist z.B. die Möglichkeit zu nennen, Filter durch Matlab testen zu lassen, indem Matlab Daten blockweise zur Eingabe an ein Filter übergibt und die entsprechende Filterausgabe analysiert.

GenMAD bietet also vielfältige Möglichkeiten zur Erweiterung, die den Rahmen möglicher Anwendungen noch einmal vergrößern können. Dies umfasst auch das Einbinden zukünftiger Technologien.

Anhang A

Hinweise zur Verwendung des Filter-SDK

Vorausgesetzt wird die Integrität der Verzeichnisstruktur, d.h. auf der obersten Verzeichnisebene `GenMAD-SDK` des SDK liegen mindestens die folgenden Unterverzeichnisse vor:

```
GenMAD-SDK
   \BaseFilter
   \DirectX
   \Interfaces
   \SDK
```

`BaseFilter` enthält für das SDK grundlegende Klassen, `DirectX` die unter Windows XP kompilierten Bibliotheken, sowie Quellen und Includes von DirectX9, `Interfaces` die öffentlichen GenMAD-Schnittstellen und `SDK` einen leeren Prototyp. Das SDK verwendet lediglich die STL, benötigt also keine MFC-Komponenten.

Um ein kompilierbares Filter zu erstellen, sind die folgenden Schritte nötig:

1) Kopieren des Verzeichnisses `SDK` in ein neues Unterverzeichnis von `GenMAD-SDK`.
2) Öffnen der kopierten Datei `Plugin1.dsw` in Visual C++ 6.0
3) Ändern der Datei `Plugin1.def`: Oben neuen Ausgabenamen spezifizieren
4) In den Projekteigenschaften (Alt-F7) diesen Namen unter *Linker➔Allgemein➔Name der Ausgabedatei* ebenfalls angeben
5) In `FilterDefs.h` muss Grundlegendes zum neuen Filter angegeben werden:
   - neue GUID (bitte **UNBEDINGT** neu generieren!!! Siehe Kommentare in Code!)
   - neuen Klassennamen und zu veröffentlichenden Filternamen angeben
6) Das SDK-Projekt enthält in `NewFilter.h/.cpp` die zu ändernde Filter-Klasse
   - ändere Klassennamen (siehe 5.)
   - definiere eigene zu veröffentlichende Variablen
   - implementiere/überschreibe Funktionen (bzw. lösche nicht verwendete Beispiel-Funktionen)
7) Nach dem Kompilieren muss die Filter-Datei (z.B. `MeinFilter.ax`) einmalig registriert werden:
   `regsvr32 MeinFilter.ax`
   (Hierbei wird auch der Pfad festgelegt, danach darf die Datei also nicht mehr verschieben! Der regsvr-Parameter /u hebt die Registrierung auf.)

Falls – entgegen der Empfehlung – ohne Verwendung der `NewFilter`-Dateien gearbeitet werden soll, ist unbedingt darauf zu achten, dass die folgenden Include-Befehle in der nachfolgend abgebildeten Reihenfolge zu oberst in der entsprechenden Header-Datei des neue Filters deklariert werden:

```
#include "..\BaseFilter\MonitoringPlugin.h"
#include "..\BaseFilter\PropertyPage.h"
#include "FilterDefs.h"
#include "..\BaseFilter\SDK.h"
```

Anhang B

Hinweise zur Verwendung von GenMAD

GenMAD wurde in Visual C++ 6.0 unter Windows XP entwickelt und auch unter Windows 2000 getestet. Der Code setzt dabei auf MFC und dem DirectX 9.0 SDK Update vom Sommer 2003 auf. Da die Änderungen des Updates die DirectShow-Komponente nicht betreffen, dürfte auch das Haupt-SDK zu DirectX 9.0 zu einer problemlosen Kompilierung führen.

GenMAD setzt zur Laufzeit eine korrekte Installation von DirectX 9 voraus. Das SDK wird nicht benötigt. Eine Installation von GenMAD ist nicht nötig, GenMad.exe ist ein direkt ausführbares Programm.

Wie alle DirectX-Filter müssen auch die GenMAD-Filter vor der Verwendung in GenMAD oder einem anderen auf DirectX aufsetzenden Programm einzeln im Zielsystem registriert werden. Dies kann durch direktes Verwenden der üblicherweise im `System32`-Unterverzeichnis von Windows befindlichen `regsvr32.exe` oder darauf aufsetzenden Hilfsprogrammen geschehen und erfordert im einfachsten Fall den folgenden Befehl:

```
regsvr32 MeinFilter.ax
```

Siehe dazu auch die Hinweise in Anhang A, Abschnitt 7.

Die bei der Kompilierung gelegentlich auftretende Warnung LNK4098 rührt aus der Verwendung von DirectShow her und kann ignoriert werden.

Anhang C

Benutzerhandbuch

GenMAD wird gestartet durch Aufruf von GenMAD.exe. Das Hauptfenster von GenMAD ist aufgebaut wie in Abbildung C.1 dargestellt.

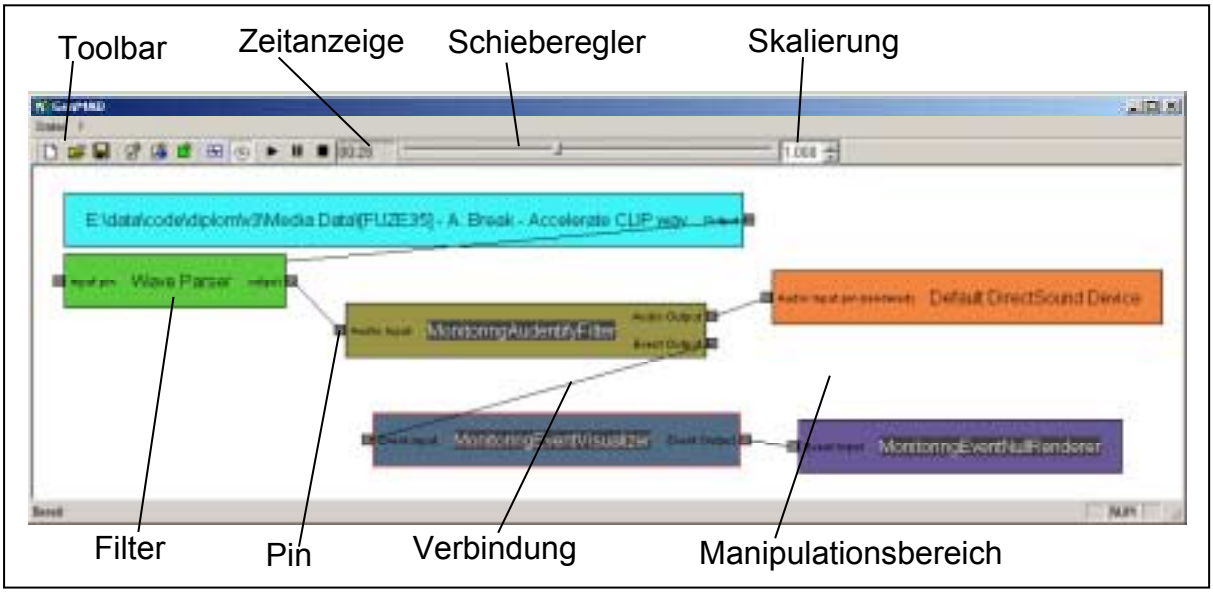


Abbildung C.1: GenMAD-Hauptfenster

Manipulationsbereich

Die im aktuellen Datenflussgraphen enthaltenen *Filter* sind als farbige Rechtecke im *Manipulationsbereich* dargestellt. Jedes Filter besitzt einen jeweils zentral angezeigten Namen und verfügt über einen oder mehrere *Pins*. Pins dienen entweder dem Eingang oder dem Ausgang von Daten zu einem Filter. Eingangs-Pins sind als kleine graue Quadrate an der linken Seite eines Filters mit nebenstehendem Namen abgebildet, Ausgangs-Pins an der rechten. *Verbindungen* zwischen Filtern bzw. Pins sind als schwarze Linien dargestellt, die sich zwischen Pins befinden. Über diese Verbindungen werden Daten von Filter zu Filter transportiert. Die im Rahmen der vorliegenden Diplomarbeit implementierten Filter werden in Abschnitt 5.3 vorgestellt.

Ein Filter lässt sich markieren, in dem man es mit der linken Maus anklickt. Das aktuell selektierte Filter wird rot umrandet dargestellt. Drückt man die Taste „Entf“ bzw. „Del“, so werden das selektierte Filter aus dem Graphen und Verbindungen zu anderen Filtern entfernt. Darüber hinaus lassen sich Filter per Drag’n’Drop verschieben. Die meisten Filter verfügen über einen Konfigurationsdialog, der sich über einen Rechtsklick auf das Filter aufrufen lässt. Klickt man einen Pin mit der rechten Maustaste an, so erscheint ein Kontextmenü mit den Einträgen „Neue Verbindung“ oder „Verbindung lösen“ (je nachdem, ob bereits eine Verbindung besteht) und „Medien-Typen anzeigen“. Der Befehl „Neue Verbindung“ ruft einen Dialog auf, der in einer Baumstruktur alle Filter mit entsprechenden Pins enthält, mit denen der gewählte Pin

verbunden werden kann. Durch Anwahl und „OK“ wird versucht, eine Verbindung herzustellen. „Verbindung lösen“ zerstört die Verbindung des gewählten Pins. Durch Aufruf von „Medien-Typen anzeigen“ wird eine dialogbasierte Liste aller unterstützter Dateiformate erfragt werden. Die Vielfalt der Informationen hängt dabei bei manchen Filtern von der Tatsache ab, ob der Pin verbunden ist oder nicht.

Toolbar

Die Toolbar umfasst eine Reihe von Buttons, die nachfolgend einzeln vorgestellt werden:

	Diese drei Buttons dient der Projektarbeit und erlauben das Erzeugen eines neuen Projekt, das Öffnen eines bestehenden Projektes und das Speichern des aktuellen Projektes.
	Mittels dieser Knöpfe können neue Filter zum Graphen hinzugefügt werden: Die ersten beiden Buttons erlauben das Hinzufügen einer lokalen bzw. einer per URL verfügbaren Mediendatei. Dabei wird automatisch eine zur Ausgabe des jeweiligen Dateityps benötigte Filterkette erzeugt. Der dritte Button, dessen Funktion auch durch Rechtsklick in den Manipulationsbereich erreicht werden kann, öffnet einen Dialog, der das manuelle Hinzufügen neuer Filter erlaubt. Dabei lassen sich mehrere Filter hintereinander hinzufügen, ohne dass der Dialog neu geöffnet werden muss.
	Dieser Button öffnet die das Visualisierungsfenster. Diese Funktion kann auch durch Doppelklicken eines Filters mit der linken Maustaste erreicht werden. In letzterem Fall wird dem Filter – falls möglich – der Audiofokus zugewiesen.
	Mittels dieses Buttons lässt sich die Synchronisation des Graphen zur Referenzuhr an- und abschalten. Standardmäßig ist sie aktiviert.
	Diese drei Buttons kontrollieren in gewohnter Weise den Datentransport.

Tabelle C.2: Buttons der Toolbar

Darüber hinaus befinden sich in der Toolbar drei weitere Steuerelemente:

- Uhr, welche die aktuelle Medienzeit anzeigt
- Schieberegler zum Vor- und Rücklauf, falls vom Graphen unterstützt
- Kontrollfeld zur Eingabe eines Skalierungsfaktors. Standardwert ist 1,0.

Visualisierungsfenster

In der *Toolbar* lassen sich drei Dinge konfigurieren:

- An- und Abschalten der Echtzeit-Darstellung (Standard ist „an“.)
- Zuweisung des Audiofokus’ an ein Filter
- Einstellen des Zoomfaktors von Audiosignal- und Event-Anzeige mit zugehörigem Feld zur Anzeige des gewählten Faktors. Der Wert definiert die Anzahl Millisekunden je Pixel. Eine Änderung des Zoomwerts wird durch einen blauen vertikalen Strick in den beiden horizontalen Anzeigen visualisiert.

Die *Audiosignal-Anzeige* stellt die Wellenform desjenigen Filters dar, dem der Audiofokus zugewiesen wurde. Diese Anzeige funktioniert nur in Echtzeit und wird ausgeblendet, falls sich die Echtzeit-Darstellung deaktiviert ist oder keinem Filter der Audiofokus zugewiesen ist. Die rot dargestellte Skala zeigt die Zeitachse an.

In der *Eventlisten-Anzeige* werden eingehende Events in Echtzeit aufgelistet. Neue Event werden dabei oben angefügt. Die Farbe eines Events entspricht dabei der Farbe desjenigen Filters im Manipulationsbereich, das den Event visualisiert hat, üblicherweise als ein Filter vom Typ *MonitoringEventVisualizer*. Die Liste enthält alle zum jeweiligen Event verfügbaren Informationen in separaten, in der Größe veränderbaren Spalten.

Auf entsprechende Weise und Farbigkeit werden in der *Event-Anzeige* Events in einer Art Piano-Roll angezeigt. Die Höhe eines als vertikaler Strich dargestellten Events hängt von seinem Score/Ranking ab. Zusammengehörige Events eines Filters werden nach einem in Abschnitt 5.1.9. beschriebenen Verfahren ermittelt und durch einen darüber liegenden horizontalen Balken dargestellt. Diese Anzeige lässt sich im abgeschalteten Echtzeit-Modus durch den darunter befindlichen Schieberegler zeitliche scrollen.

Die Größenverhältnisse der drei Anzeigen können durch Drag'n'Drop der Splitter verändert werden.

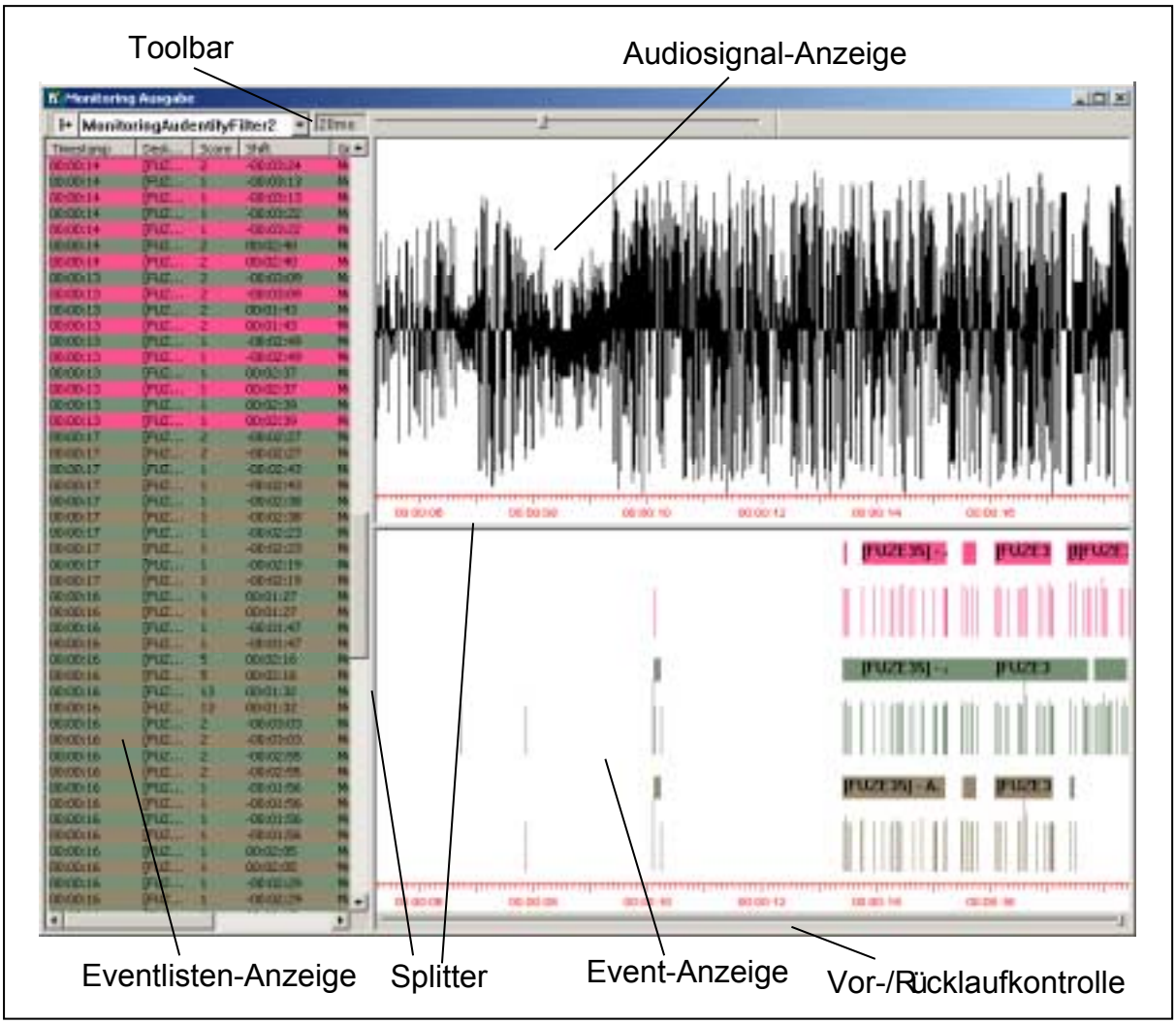


Abbildung C.3: Das Visualisierungsfenster

Arbeitshinweise

Es empfiehlt sich, zum Aufbau eines Datenflussgraphen in GenMAD analog zum Fluss des Graphen vorzugehen:

Zunächst fügt man durch Anklicken der entsprechenden Buttons in der Toolbar eine oder mehrere neue Mediendateien zum Graphen hinzu. Dabei werden jeweils automatisch auch diejenigen Filter erzeugt und verbunden, die zur Transformation und Ausgabe des entsprechenden Dateityps nötig sind. Da manche Filtertypen erst bei korrekter Parametrisierung bestimmte Pins veröffentlichen, sollte jedes Filter sofort nach dem Einfügen konfiguriert werden. Diesen grundlegenden Graphen erweitert man nun durch zusätzliche Filter, die ebenfalls – wie oben erläutert – mittels der Toolbar hinzugefügt und manuell an geeigneter Stelle in den Signalfluss eingesetzt werden, indem eine existierende Verbindung zweier Filter entfernt wird und die Pins des einzubindenden Filters mit den nun freien Pins verbunden werden.

Wenn der Graph komplett erstellt, konfiguriert und verbunden wurde, kann der Datenfluss durch Starten des Graphen initiiert werden. Der Aufruf des Visualisierungsfensters kann auch bei laufendem Graphen erfolgen. Das Hinzufügen neuer Filter zur Laufzeit wird allerdings nicht empfohlen – dazu sollte sich der Graph in gestopptem Zustand befinden.

Anhang D

Technische Dokumentation von GenMAD

Diese Dokumentation soll vor allem dazu dienen, eine effektive Weiterentwicklung des GenMAD-Systems zu ermöglichen und das Verständnis für die Funktionsweise von GenMAD zu vertiefen, vor allem in Verbindung mit dem Quellcode. Der Fokus liegt daher auf denjenigen Schnittstellen und Klassenelementen, die wesentliche Teile des Systems darstellen oder Möglichkeiten zur Erweiterung bieten. Triviale Klassen und Methoden mit eingeschränkter Funktionalität werden daher nicht bis ins letzte Detail, sondern nur rudimentär erläutert. Für eine umfassende Dokumentation auch dieser Elemente sei daher auf die im Quellcode enthaltene Dokumentation verwiesen.

Ergänzende Hinweise zum Zusammenspiel der jeweiligen Klassen finden sich in den Kapiteln 4 und 5, zusammen mit mehreren Abbildungen.

Es sei darauf hingewiesen, dass ein Rückgabewert vom Typ HRESULT einen Integerwert liefert, der den Erfolg der Operation bewertet. Dabei kennzeichnet generell ein Wert ungleich S_OK einen Fehler. Dieses Vorgehen ist in DirectShow üblich und in GenMAD soweit möglich und sinnvoll ebenfalls implementiert.

Dieser Teil des Anhangs ist in vier Bereiche gegliedert: Zunächst folgt in D.1 ein Überblick über das Zusammenspiel der einzelnen Komponenten. In den nachfolgenden Kapiteln D.2 bis D.4 folgen Erläuterungen der in GenMAD enthalten Klassen, globalen Schnittstellen und Strukturen.

D.1 Überblick

In Abbildung D.1 sind alle GenMAD-Klassen in einem UML-Diagramm dargestellt.

D.2 Klassen

Nachfolgend werden alle Klassen von GenMAD dargestellt.

D.2.1 CAboutDlg

`class CAboutDlg : public CDialog`
MFC-Standarddialog mit Copyright-Informationen, der bei Aufruf des einzigen Menüpunkts „*Info über GenMAD*“ im Hilfemenü angezeigt wird.

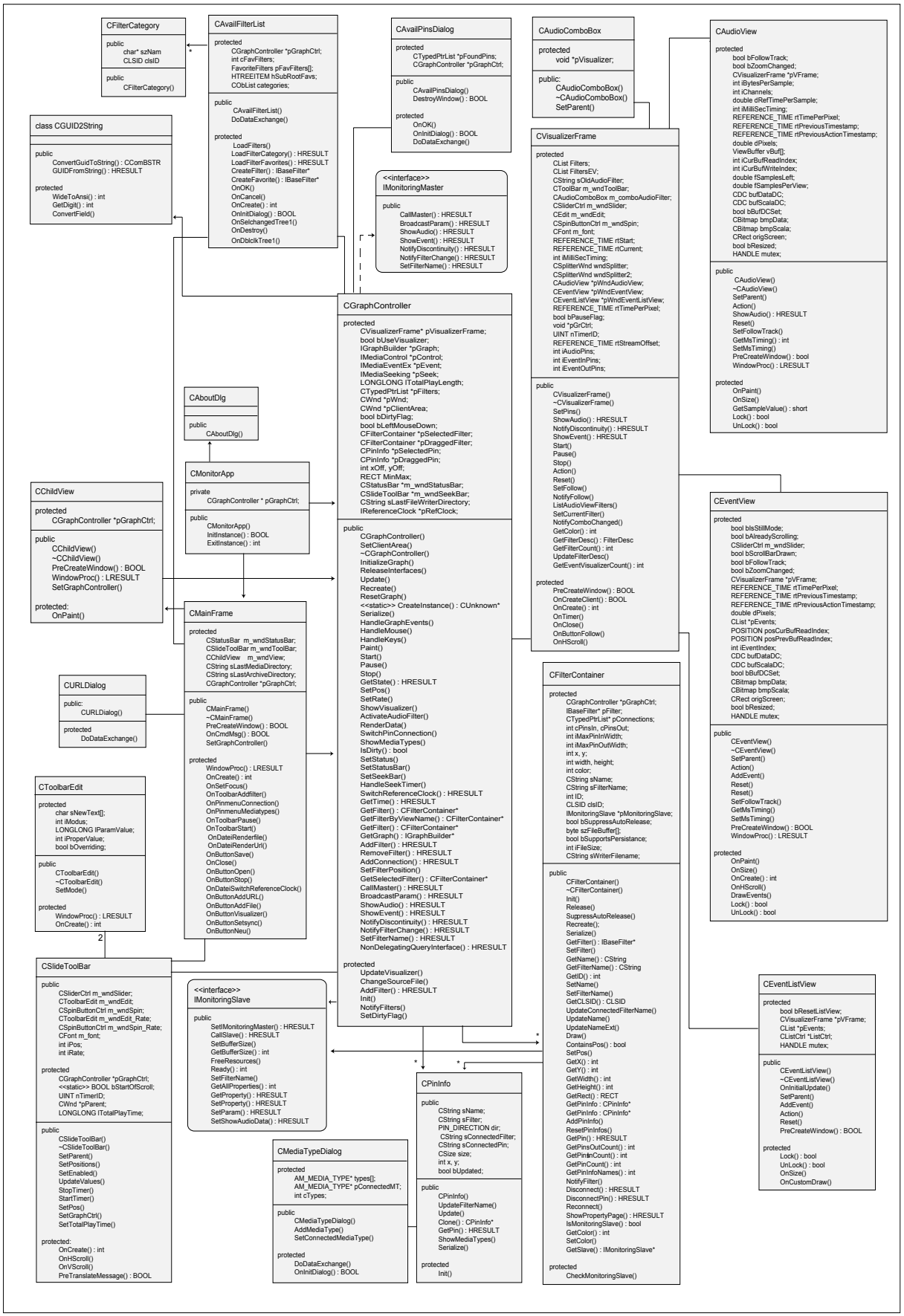


Abbildung D.1: UML-Diagramm von GenMAD

D.2.2 CAudioComboBox

`class CAudioComboBox : public CComboBox`
Simple ComboBox, die im Visualisierungsfenster das Filter mit dem aktuellen Audiofokus verwaltet. Bei Benutzer-Auswahl eines neuen Filters, angezeigt durch die Windows-Message `CBN_SELENDOK` wird `CVisualizerFrame::NotifyComboChanged()` aufgerufen.

D.2.3 CAudioView

`class CAudioView : public CWnd`
Diese Klasse ermöglicht die Visualisierung von Audiodaten. Dazu werden aus eingehenden Audiodaten Extremwerte extrahiert und in einem ringförmigen Datenpuffer abgelegt:

```
ViewBuffer vBuf[VBUFSIZE];    //Puffer für extrahierte Audiodaten-Extrema
int iCurBufReadIndex;
int iCurBufWriteIndex;        //Lese-/Schreibindizes
```

Das Zeichnen erfolgt über versteckte Zeichenpuffer:

```
CBitmap bmpData, bmpScala;    //versteckte Zeichenpuffer für Daten und Skala
CDC bufDataDC, bufScalaDC;    //Zeichenobjekte für den Zugriff darauf
bool bBufDCSet;               //Flag: Wurden die Puffer erzeugt?
CRect origScreen;             //Umfang des tatsächlichen Ausgabebereiches
```

`void OnPaint()`
Diese Methode erzeugt und initialisiert zunächst bei Bedarf neue versteckte Zeichenpuffer und übertragt im Anschluss deren Inhalt skaliert auf den sichtbaren Zeichenbereich des Fensters, um so Flackern zu vermeiden.

`void Action(REFERENCE_TIME rtNow)`
In dieser Methode werden anhand der Audiodaten-Extrema, des vorigen Zeitstempels und der übergebenen aktuellen Zeit die Puffer um einige Pixel verschoben und der entsprechende freie Bereich neu beschrieben. Der Aufruf erfolgt dabei Timer-gesteuert aus CVisualizerFrame heraus.

`HRESULT ShowAudio(AudioBuffer ab)`
Dieser Funktion werden anzuzeigende Audiodaten übergeben, aus denen anhand der aktuellen Zoomeinstellung Extremwerte extrahiert und diese im Datenpuffer abgelegt werden.

`short GetSampleValue(AudioBuffer ab, PBYTE pSample)`
Hilfsfunktion, liefert unter Verwendung von Informationen zum aktuellen Audiodatentyp ein Sample an übergebener Speicherstelle `pSample`.

`void Reset()`
Setzt Variablen und Audiodatenpuffer zurück und zerstört die Zeichenpuffer.

```
HANDLE mutex;
bool Lock()
bool UnLock()
```

Mittels des Handles und der beiden Funktionen wird der Zugriff innerhalb der Klasse serialisiert, vor allem bezüglich des Zugriffs auf den Datenpuffer durch mehrere asynchrone Threads. Das Handle selbst wird im Klassenkonstruktor erzeugt und im Destruktor zerstört.

D.2.4 CAvailFilterList

`class CAvailFilterList : public CDialog`
Dieser Dialog beinhaltet ein Baum-Steuerelement, das bei Initialisierung mit den kategorisierten DirectX-Filtern gefüllt wird, die im System registriert sind. Durch Klicken auf einen Button wird in void `OnOK()` ein Filter des ausgewählten Typs instanziiert und dem Graphen hinzugefügt. Die Identifikationsinformationen jedes aufgeführten Filters werden zum Erzeugen dadurch ermittelt, dass der zum Füllen des Baums verwendete Algorithmus ein zweites Mal, und zwar bis zu dem Index, der zum gewählten Filter gespeichert wurde, durchlaufen wird. Das Erzeugen selbst erfolgt über eine DirectShow-Methode in `CreateFavorite()`.

Die Kategorien sind dabei als `CFilterCategory`-Objekte einer CObList fix vordefiniert:
```
CObList categories;
```

Ebenso existiert ein Feld `pFavFilters` mit vordefinierten Filtern, das während des Auflistungsprozesses in `LoadFilters()` dynamisch um Filter mit dem Namenspräfix *Monitoring* ergänzt wird.
```
int cFavFilters;
FavoriteFilters pFavFilters[MAX_FAVS_COUNT];
```

Wesentliche Methoden sind:

`CAvailFilterList(CGraphController *pGraphController, CWnd* pParent)`
Definition von Favoriten und Kategorien.

`void OnOK()`
Aufruf der erzeugenden Methoden CreateFavorite() bzw. CreateFilter() unter Einbeziehung von Informationen des aktuell gewählten Baumknotens, falls ein Filter gewählt ist.

`void LoadFilters()`
Erzeugen des Baum-Steuerelementes, sukzessives Aufrufen von `LoadFilterFavorites()` bzw. `LoadFilterCategory()`.

`HRESULT LoadFilterFavorites(CFilterCategory *pCat, POSITION pos)`
Einfügen der in `pFavFilters` gespeicherten und als `pCat` übergebenen Filter in den Baum. Dabei wird der Zweig des Baumes mit dem Kategorie-Index `pos` markiert.

`HRESULT LoadFilterCategory(CFilterCategory *pCat, POSITION pos)`
Unter Verwendung eines COM-Objektes namens SystemDeviceEnumerator wird eine Filter-Kategorie `pCat` im System erfragt und in den Baum eingefügt. Der Zweig des Baumes wird mit dem Kategorie-Index `pos` markiert. Der Vorgang wird in 5.1.3 zusammengefasst.

`IBaseFilter* CreateFavorite(int iIndex)`
Erzeugung eines als Favorit definierten Filters an Stelle `iIndex` im vordefinierten Feld, das als `IBaseFilter` zurückgegeben wird.

`IBaseFilter* CreateFilter(CLSID categoryCLSID, int iIndex)`
Erzeugung eines durch Filters der Kategorie `categoryCLSID` an Stelle `iIndex` im vordefinierten Feld mittels des beschriebenen Vorgangs. Das Filter wird als `IBaseFilter` zurückgeliefert.

D.2.5 CAvailPinsDialog

Dialog, der zu einem gegebenen Pin in einer Baumstruktur alle Pins mit zugehörigem Filter anzeigt, die sich mit einander verbinden ließen. Auf Knopfdruck wird dies ausgeführt.
Die verbindbaren Pins werden dem Dialog übergeben und in einer CList gespeichert:
`CTypedPtrList <CPtrList, CPinInfo*> *pFoundPins;`

`void OnOK()`
Anhand des gewählten Eintrags im Baum-Steuerelement wird versucht eine Verbindung dazu herzustellen. Im Erfolgsfall wird der Dialog geschlossen.

D.2.6 CChildView

`class CChildView : public CWnd`
Diese Klasse repräsentiert den Manipulationsbereich und dient einzig dem Zweck, von Windows versandte Zeichnen- und Benutzereingabe-Messages abzufangen und an `CGraphController` weiterzureichen.

D.2.7 CEventListView

`class CEventListView : public CListView`
CListView-Klasse des Visualisierungsfensters, das eingehende Events in einer CList speichert und mittels eines CListCtrl-Steuerelements anzeigt:
`CList<EventSample, EventSample&> *pEvents;`

Dabei wird ebenfalls ein in D.2.3 besprochenes Mutex-Handle zur Serialisierung verwendet.

`void AddEvent(EventSample es)`
Fügt neue Events am Ende von `pEvents` an.

`void Action(REFERENCE_TIME rtNow)`
Einfügen aller in `pEvents` vorhandenen Events in das ListCtrl-Objekt unter Verwendung der übergebenen aktuellen Zeit. Abschließend wird `pEvents` geleert.

`void OnCustomDraw(NMHDR* pNMHDR, LRESULT* pResult)`
Jede Zeile wird entsprechend der Farbe des visualisierenden Filters mittels eines zweistufigen Prozesses koloriert.

D.2.8 CEventView

`class CEventView : public CWnd`
In ähnlicher Weise wie CEventListView werden auch in dieser Klasse eingehende Events durch `AddEvent()` in einer CList gespeichert und separat angezeigt, hier in Form eines horizontalen *Piano Roll*.
Allerdings wird bei der Speicherung jedem Events noch ein Flag hinzugefügt, das Auskunft darüber gibt, ob ein Event bereits angezeigt wurde:
`CList<EventSampleWrapper, EventSampleWrapper&> *pEvents;`

Das in D.2.3 erläuterte Konzept versteckter Zeichenpuffer wird hier ebenso identisch angewandt wie das der Zugriffsserialisierung.

`void DrawEvents(REFERENCE_TIME rtNow, bool bStillMode)`
Wie in 5.1.9 erläutert werden hier Events separat nach visualisierenden Filtern und in Sub-Bahnen gleicher Eventbeschreibungen angezeigt. Die Parameter geben Auskunft über die aktuelle Zeit und den Status, ob die Anzeige in Echtzeit verwendet wird oder über den in der Klasse enthaltenen Schieberegler verändert wurde:
`CSliderCtrl m_wndSlider;`
Beim Anzeigen bedient sich die Funktion mehrerer Methoden der `CVisualizerFrame`-Klasse, die Informationen zu den verschiedenen Filtern verwaltet und aktualisiert, z.B. zu Eventblocks und dem vorigen Event.

`void AddEvent(EventSample es)`
Die übergebenen Events werden im Gegensatz zu `CEventListView` in `pEvents` sortiert eingefügt, wobei ein Zeiger auf das nächste zu zeichnende Event ggf. aktualisiert wird.

D.2.9 CFilterCategory

`class CFilterCategory : public CObject`
Hilfsklasse zu CAvailFilterList, die das Speichern vordefinierter Filterkategorien in einer CObList ermöglicht. Die einzigen Elemente sind zwei Variablen und ein diese parametrisierender Konstruktor:

```
char* szNam;      //Kategorie-Name
CLSID clsID;      //Kategorie-CLSID
```

D.2.10 CFilterContainer

`class CFilterContainer : public CObject`
Repräsentantenklasse eines Filters, die über Methoden zur Host-Filter-Kommunikation, zur Übergabe von Informationen und zur Manipulation von Filter-Eigenschaften verfügt.
Wichtige Eigenschaften:

```
CTypedPtrList<CPtrList,CPinInfo*>* pConnections;     //Pins
IBaseFilter* pFilter;   //das repräsentierte DirectX-Filter
int cPinsIn, cPinsOut;  //Anzahl Pins in / out
CString sName;          //interner name
CString sFilterName;    //angezeigter Name, z.b. nach manuellem Umbenennen
CLSID clsID;            //CLSID des Filters
IMonitoringSlave *pMonitoringSlave; //Zeiger auf Interface, evtl. NULL
```

Folgende Methoden sind wesentliche Bestandteile der Klasse:

`CFilterContainer(CGraphController *pGrCtrl, IBaseFilter *pFlt)`
Übergeben werden diesem Konstruktor Zeiger auf CGraphController und das dieser Instanz zugrunde liegende Filter. Nach Aufruf der in `Init()` zentral ausgeführten Initialisierung werden Informationen durch Abfrage des Filters eingeholt und gespeichert.

`void Draw(CClientDC *pDC)`
Anhand der Filtereigenschaften wird in dieser Methode der übergebene Zeichenbereich beschrieben.

`void NotifyFilter(CClientDC *pDC)`
Durch diese Methode wird die Klasse über Änderungen informiert, z.B. nach Herstellen einer neuen Verbindung. Dabei werden sämtliche Graph-bezogenen äußeren Informationen des Filters aktualisiert. Der übergebene Zeichenbereich wird dazu verwendet, die Ausdehnung von Beschriftungen zu ermitteln und für eine schnellere Ausführung von `Draw()` zu speichern.

`CPinInfo* GetPinInfo(CString szNam)`
Liefert zu einem Pin-Namen das entsprechende CPinInfo-Objekt zurück, also die GenMAD-Repräsentation eines Pins.

`HRESULT GetPin(CString szQueriedName, IPin** ppPin)`
Liefert zu einem Pin-Namen das entsprechende IPin-Objekt, also das DirectShow-Objekt, zurück.

`HRESULT Disconnect()`
Entfernt alle Verbindungen zu benachbarten Filtern.

`HRESULT DisconnectPin(CPinInfo *pPinInfo)`
Entfernt die Verbindung des übergebenen Pins.

`void Reconnect(bool bRecursive)`
Gemäß `pConnections` alle dort gespeicherten Verbindungen rekursiv wieder herstellen, z.B. nach Laden des Graphen aus einer Datei.

`void UpdateConnectedFilterName(CString sNameOld, CString sNameNew)`
`void UpdateName(CString sNameOld, CString sNameNew)`
Diesen beiden Methoden dienen der automatischen Anpassung von Filternamen
`void UpdateNameExt(CString sNameOld, CString sNameNew)`
Entsprechend im Fall manueller Anpassung

`void Serialize(CArchive& archive)`
Wie in Abschnitt 5.1.4 beschrieben findet hier das Laden und Speichern mittels des CArchive-Objektes statt.

`void Recreate(CGraphController *pGrCtrl)`
Neu-Erzeugen des Filters unter Einbeziehen interner Eigenschaften, z.B. nach Ladevorgang

`HRESULT ShowPropertyPage()`
Konfigurationsdialog des Filters aufrufen

D.2.11 CGraphController

```
class CGraphController : public CObject, public CUnknown,
                                                public IMonitoringMaster
```

Dies ist die zentrale Klasse des GenMAD-Systems. Wie in Kapitel 4 erläutert repräsentiert sie das `GraphBuilder`-Objekt, das in DirectShow den Datenflussgraphen kontrolliert.

Zentrale Eigenschaften:

```
CVisualizerFrame* pVisualizerFrame; //Visualisierungsfenster

//zentrale DirectShow-Handles
IGraphBuilder *pGraph;              //GraphBuilder
IMediaControl *pControl;            //Abspielkontrolle
IMediaEventEx *pEvent;              //Ereignisbehandlung
IMediaSeeking *pSeek;               //Seeking

LONGLONG lTotalPlayLength;          //Gesamtspielzeit d. aktuellen Mediums

CTypedPtrList<CPtrList,CFilterContainer*> *pFilters; //die Filter
CWnd *pWnd;                         //Applikationsrahmen
CWnd *pClientArea;                  //Manipulationsbereich
IReferenceClock *pRefClock;         //Referenz-Uhr
```

Nachfolgend sind die wichtigen Methoden der Klasse aufgeführt:

`void ResetGraph()`
Zurücksetzen des Graphen: Entfernen aller Filter

`void InitializeGraph()`
Erzeugen des Graphen, Initialisieren der zentralen DirectShow-Objekte und deren Parameter

`void RenderData(CString sSource)`
Eine Mediendatei an übergebenem Pfad renderbereit zum Graphen hinzufügen, wie in 5.1.5 beschrieben.

`void HandleGraphEvents()`
DirectShow-Ereignisse behandeln, wie in 5.1.2 erläutert.

`void NotifyFilters()`
Alle Filter zum Überprüfen von Änderungen auffordern.

`void ReleaseInterfaces()`
Freigabe aller Interfaces und DirectShow-Objekte

```
HRESULT CallMaster(int iCodeID, void* pParam, void** ppReturnValue);
HRESULT BroadcastParam(int iParamID, double dValue);
HRESULT ShowAudio(AudioBuffer ab);
HRESULT ShowEvent(EventSample es);
HRESULT NotifyDiscontinuity(REFERENCE_TIME rtTimestamp);
HRESULT NotifyFilterChange(int iParam);
HRESULT SetFilterName(char *sOldName, char *sNewName);
```
Implementation des IMonitoringMaster-Interfaces, siehe dessen Beschreibung in D.3.1

`STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void ** ppv)`
DirectShow-Methode zur Veröffentlichung implementierter Schnittstellen, damit dieser via COM verfügbar sind.

`void HandleMouse(int iEvent, WPARAM wParam, LPARAM lParam)`
Behandlung von Mausereignissen im Manipulationsbereich.

`void ShowVisualizer(CFilterContainer *pFlt)`
Anzeigen des Visualisierungsfenster, Übergabe von Filter-Informationen

`void HandleKeys(WPARAM wParam, LPARAM lParam)`
Behandlung von Tastatureingaben

`void Paint()`
Zeichnen des aktuellen Graphen durch Aufruf entsprechender Methoden aller Filter

`CFilterContainer* GetFilter(int ID)`
`CFilterContainer* GetFilter(CString szName)`
`CFilterContainer* GetFilterByViewName(CString szName)`
Filter anhand von ID, internem oder angezeigtem Namen finden

`HRESULT AddFilter(CFilterContainer *pFltContainer)`
`HRESULT AddFilter(IBaseFilter *pFilter, CString sName)`
Hinzufügen eines Filters zum Graphen, entweder als Hilfsfunktion zur `RenderData()` oder nach Auswahl im `CAvailsFiltersList`-Dialog

`HRESULT RemoveFilter(CFilterContainer *pFltContainer)`
Komplettes Entfernen eines Filters aus dem Graphen

`void SwitchPinConnection()`
Umkehren des Verbindungszustandes des selektierten Pins, also *verbunden* und *nicht verbunden*.

`void ShowMediaTypes()`
Anzeige der Mediaformate eines Pins per `CMediaTypeDialog`.

`HRESULT AddConnection(IBaseFilter *pFilter)`
Verbindung zwischen selektiertem und übergebenem Filter herstellen.

`void Start()`
`void Pause()`
`void Stop()`
Abspielkontrolle, wie in 5.1.8 beschrieben. Dabei wird sowohl `pControl` verwendet als auch das Visualisierungsfenster synchronisiert.

`void Serialize(CArchive& archive)`
Wie in Abschnitt 5.1.4 beschrieben findet hier das Laden und Speichern mittels des CArchive-Objektes statt.

`void Update()`
Benachrichtigen aller Filter über mögliche Änderungen und anschließendes Zeichnen

`void Recreate(CString sOpenPath)`
Nach einem Ladevorgang ausgeführte Methode zur Verbindung aller Filter, wie in 5.1.4 erläutert.

`HRESULT SwitchReferenceClock(bool bUseReference)`
Referenz-Clock von NULL – wie per positivem Parameter angezeigt – auf normal & umgekehrt umschalten. Siehe 5.1.7 für eine Beschreibung des Konzeptes.

`void ChangeSourceFile(CFilterContainer *pFltContainer)`
Austausch einer Mediendatei eines FileSource-Filters. Wird erreicht durch Neu-Erzeugen und geeignetes Verbinden anstelle des bisherigen Filters.

`void ActivateAudioFilter(CString sName, bool bActive)`
Benachrichtigung des benannten Filters über Erhalt oder Abgabe des Audiofokus'

D.2.12 CGUID2String

`class CGUID2String`
Hilfsklasse zum Konvertieren von GUIDs aus/in Strings.

D.2.13 CMainFrame

`class CMainFrame : public CFrameWnd`
Repräsentiert das Hauptfenster.

Wichtige Eigenschaften:
```
CStatusBar  m_wndStatusBar;   //Statuszeile
CSlideToolBar m_wndToolBar;   //Toolbar
CChildView    m_wndView;      //Manipulationsbereich
```

Darüber hinaus sind Handlermethoden für die Buttons der Toolbar vorhanden, die allerdings nur Vorarbeit leisten und auf in CGraphController implementierte Funktionen verweisen.

D.2.14 CMediaTypeDialog

`class CMediaTypeDialog : public CDialog`
Dieser Dialog zeigt in einer Auflistung die separat übergebenen unterstützten Medienformate eines Pins.

D.2.15 CMonitorApp

`class CMonitorApp : public CWinApp`
Dies ist die Windows-Applikationsklasse, deren einzige Aufgabe die Erzeugung und Anzeige der CMainFrame- und der CGraphController-Klasse ist.

D.2.16 CPinInfo

`class CPinInfo : public CObject`
Dieser Klasse repräsentiert ein Pin und besitzt öffentliche Variablen für alle wichtigen Pin-Eigenschaften:

```
CString sName;                //Name des Pins
CString sFilter;              //Name des Filters
PIN_DIRECTION dir;            //Ein- oder Ausgangspin?
CString sConnectedFilter;     //Name des verbundenen Filters
CString sConnectedPin;        //Name des verbundenen Pins
```

Im Konstruktor werden alle Eigenschaften durch Abfrage des Pins ermittelt. Wie auch in `CGraphController` und `CFilterContainer` wird hier das Laden und Speichern über `CArchive` bzw. `Serialize()` unterstützt. Da vorausgesetzt, ist `CPinInfo` daher von `CObject` abgeleitet und enthält wie diese zwei notwendige Makros.

`HRESULT GetPin (CGraphController *pGraphCtrl, IPin **ppPin)`
Liefert ein IPin-Objekt des repräsentierten Pins zurück.

`void ShowMediaTypes(CGraphController *pGraphCtrl)`
Ermittelt und zeigt unterstützte Mediaformate in einem CMediaTypeDialog an.

D.2.17 CSlideToolBar

`class CSlideToolBar : public CToolBar`
Diese Toolbar-Klasse beinhaltet neben einigen Buttons auch einen Schieberegler mit zugehöriger Anzeige (genannt *Seekbar*), mittels der sich – falls vom aktuellen Graphen unterstützt – ein Vor- und Rücklauf durchführen lässt. Dazu werden die folgenden drei Steuerelemente verwendet:

```
CSliderCtrl m_wndSlider;          //Schieberegler
CToolbarEdit m_wndEdit;           //Anzeige
CSpinButtonCtrl m_wndSpin;        //Editierung, unsichtbar
```

Diese werden über einen Timer `nTimerID` in Millisekundenabständen aktualisiert.

Darüber hinaus ist ein Element zur Kontrolle der Skalierung enthalten:

```
CToolbarEdit m_wndEdit_Rate;        //Anzeigefeld
CSpinButtonCtrl m_wndSpin_Rate;     //Editierung, sichtbar
```

Bei Veränderung der Steuerelemente werden die neuen Werte ausgelesen und an CGraphController weitergereicht.

D.2.18 CToolbarEdit

```
class CToolbarEdit : public CEdit
```

Dieses Steuerelement zeigt andere Werte an als intern gehalten werden und findet Verwendung in Unterstützung der Slider- & SpinCtrl in der Seekbar von CSlideToolBar. Die internen Werte werden anhand eines von z.Zt. zwei festgesetzten Modi auf andere Werte abgebildet, in dem Windows-Nachrichten abgefangen werden. Dies umfasst auch Benutzereingaben.

```
//Zeit-Anzeige, generiert aus 100ns-Einheit Stunden:Minuten:Sekunden
#define TOOLBAREDIT_MODE_TIME 0
//Skalierungs-Anzeige, zeigt ein Tausendstel des tatsächlichen Wertes an
#define TOOLBAREDIT_MODE_1000 1
```

D.2.19 CURLDialog

```
class CURLDialog : public CDialog
```

Hierbei handelt es sich um einen simplen Dialog zur Eingabe einer URL.

D.2.20 CVisualizerFrame

```
class CVisualizerFrame : public CFrameWnd
```

Diese Klasse repräsentiert das Visualisierungsfenster. Es enthält dazu Instanzen der drei Teilbereiche, die zusammen mit trennenden CSplitter-Steuerelementen erzeugt werden.

```
CAudioView *pWndAudioView;          //Audioanzeige
CEventView *pWndEventView;          //Event-Piano-Roll
CEventListView *pWndEventListView;  //Event-Liste
```

Informationen über im Graph enthaltene Filter werden mit weiteren Angaben angereichert und als `CList` von `FilterDesc`-Objekten lokal gespeichert:

```
CList<FilterDesc, FilterDesc&> Filters;  //Liste aller Generatoren
CList<FilterDesc, FilterDesc&> FiltersEV;//Liste der visualizerenden Filter
```

Darüber hinaus verwendet die Klasse einen Timer, um in Millisekundenabständen die Teilbereiche zum Aktualisieren aufzufordern.

```
HRESULT ShowAudio(AudioBuffer ab)
HRESULT ShowEvent(EventSample es)
```

Diese beiden Methoden erhalten zu visualisierende Daten und leiten sie an die entsprechenden Teilbereiche weiter.

```
void Start(REFERENCE_TIME rtNow);
void Pause();
void Stop();
```

Diese drei Abspielfunktionen werden von CGraphController synchronisiert und kontrollieren den Timer. Dieser wiederum ruft die folgende Methode auf:

```
void Action()
```
Diese Methode ruft gleichlautende Funktionen in allen drei Teilbereichen auf.

```
void Reset()
void Reset(bool bAudioOnly)
```
Diese Funktionen setzen die Visualisierungs-Anzeige zurück, z.B. wenn der Graph gestoppt wird. Das `bAudioOnly`-Flag kontrolliert das Zurücksetzen der Filterinformationen und wird im Zusammenhang mit dem Umschalten zwischen Echtzeit- und Standbyanzeige eingesetzt.

```
void ListAudioViewFilters(CList<FilterDesc,FilterDesc&>* pFilters,
                          CList<FilterDesc,FilterDesc&>* pFiltersEV)
```
Dieser Methode werden Informationen zu Generator- und visualisierenden Eventfiltern übergeben. Daraufhin werden die Bahnen der Piano-Roll-Anzeige ggf. neu vergeben und die Auswahlbox des Audiofokus überprüft.

```
void OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
```
Diese Methode zur Ereignisbehandlung korrespondiert mit dem Zoom-Schieberegler und übergibt den Zoomwert den entsprechenden Teilbereichen.

```
FilterDesc GetFilterDesc(CString sName)
int GetFilterCount()
void UpdateFilterDesc(FilterDesc fd)
```
Diese drei Methoden befassen sich mit der Bereitstellung von Filterinformationen an die Teilbereiche und liefern zu einem übergebenen Filternamen ein `FilterDesc`-Objekt, geben die Anzahl vorhandener Einträge zurück und aktualisieren vorhandene Einträge.

D.2.21 DShowUtils

Abschließend sei auf folgenden Namespace hingewiesen, der eine Reihe von DirectShow-nahen Funktionen enthält, die von verschiedenen GenMAD-Klassen aufgerufen werden. Diese Funktionen entstammen teilweise dem DirectX-SDK oder sind Erweiterungen davon, andere wurden vom Autor dieser Diplomarbeit implementiert. In der folgenden Auflistung wichtiger enthaltener Funktionen sind – durch *(..)* gekennzeichnet – die Parameter absichtlich weggelassen, um die Lesbarkeit zu erhöhen. Diese sind jedoch wie gewohnt im Quelltext dokumentiert:

```
CString GetDirectXErrorMessage(HRESULT hr);
```
Liefert zu einem Fehlercode die zugehörige Fehler-Meldung.

```
char* GetMediaType(GUID mediatype);
```
Liefert zum GUID eines DirectShow-Medienformates den zugehörigen Bezeichner.

```
HRESULT CreateFilterByCLSID(..)
```
Durch CLSID definiertes Filter erzeugen.

```
HRESULT AddFilterByCLSID(..)
```
Durch CLSID definiertes Filter zum Graphen hinzufügen.

```
HRESULT ConnectFilters(..)
```
Verbindet zwei Filter, einer davon gegeben durch einen Pin.

```
HRESULT ConnectFilters(..)
```
Verbindet zwei Filter.

`HRESULT GetPin(..)`
Liefert zu einem Filter einen enthaltenen Pin mit vorgegebener Orientierung.

`HRESULT FindInterfaceAnywhere(..)`
Liefert ein vorgegebenes Interface, falls im Graphen implementiert.

`void ListFilters(..)`
Liefert eine Collection aller Filter im Graphen.

`void FindMatchingPins(..)`
Liefert zu einem Pin sämtliche aufgrund des Media Formates passendes Pins als Collection.

D.3 Globale Schnittstellen

D.3.1 IMonitoringMaster

DirectShow-Schnittstelle, die von GenMAD implementiert wird und Filtern den Aufruf von darin enthaltenen Funktionen ermöglicht

```
DECLARE_INTERFACE_(IMonitoringMaster, IUnknown)
{
public:
      //erweiterbare Funktionalität
      //Parameter:
      //iCodeID: Codes für generische Unterfunktionen
      //pParam: Zeiger für beliebige Parameter
      //ppReturnValue: Zeiger auf Zeiger für beliebige Rückgabewerte
      virtual HRESULT STDMETHODCALLTYPE CallMaster(int iCodeID, void*
                                    pParam, void** ppReturnValue) = 0;

      //Inter-Filter-Kommunikation
      //dient der Abstimmung von Parameterwerten
      //Parameter:
      //iParamID: ID des Parameters
      //dValue: Wert des Parameters
      virtual HRESULT STDMETHODCALLTYPE BroadcastParam(int iParamID, double
                                                       dValue) = 0;

      //Visualisierung von Audiodaten
      //Parameter:
      //anzuzeigender Audiodatenblock
      virtual HRESULT STDMETHODCALLTYPE ShowAudio(AudioBuffer ab) = 0;

      //Visualisierung von Eventdaten
      //Parameter:
      //anzuzeigender Event
      virtual HRESULT STDMETHODCALLTYPE ShowEvent(EventSample es) = 0;

      //Benachrichtigung über Unterbrechung im Stream
      //Parameter:
      //rtTimestamp: Zeitstempel
      virtual HRESULT STDMETHODCALLTYPE NotifyDiscontinuity(REFERENCE_TIME
                                                  rtTimestamp) = 0;

      //Benachrichtigung über Filter-Veränderung
      //Parameter:
      //iParam: Parameter-ID
```

```
      virtual HRESULT STDMETHODCALLTYPE NotifyFilterChange(int iParam) = 0;

      //Übergabe des Filternamens, z.B. nach Laden eines Projektes
      //Parameter:
      //sOldName: alter Name
      //sNewName: neuer Name
      virtual HRESULT STDMETHODCALLTYPE SetFilterName(char *sOldName, char
                                                      *sNewName) = 0;
};
```

D.3.2 IMonitoringSlave

DirectShow-Schnittstelle, deren Implementation Filtern die Kommunikation mit eigenen Erweiterungen, anderen Filtern und dem GenMAD-Host ermöglicht

```
DECLARE_INTERFACE_(IMonitoringSlave, IUnknown)
{
public:

      //Übergabe eines Zeigers auf den Host
      //Parameter:
      //Zeiger auf COM-Objekt, das ImonitoringMaster implementiert
      virtual HRESULT STDMETHODCALLTYPE SetIMonitoringMaster(
                                        IMonitoringMaster *pam) = 0;

      //erweiterbare Funktionalität
      //Parameter:
      //iCodeID: Codes für generische Unterfunktionen
      //pParam: Zeiger für beliebige Parameter
      //ppReturnValue: Zeiger auf Zeiger für beliebige Rückgabewert
      virtual HRESULT STDMETHODCALLTYPE CallSlave(int iCodeID, void*
                                  pParam, void** ppReturnValue) = 0;

      //Festlegen und Abfrage der Puffergrösse in Bytes
      void STDMETHODCALLTYPE SetBufferSize(int iBytes);
      int STDMETHODCALLTYPE GetBufferSize();

      //Eigene Resourcen wieder freigeben
      virtual void STDMETHODCALLTYPE FreeResources() = 0;

      //Statusabfrage, ob bereit zu starten.
      //Parameter:
      //szErrMsg: optionaler Fehlertext zur Rückgabe
      //Rückgabe: Ein Wert == 0 signalisiert Bereitschaft
      virtual int STDMETHODCALLTYPE Ready(char *szErrMsg) = 0;

      //Festsetzen des Namens
      //Parameter:
      //sNewName: Name
      virtual void STDMETHODCALLTYPE SetFilterName(char* sNewName) = 0;

      //Abfrage einer Liste aller Properties
      //Parameter:
      //ppPropertyVarType: Array mit Properties
      //Rückgabe: Anzahl der Properties
      virtual int STDMETHODCALLTYPE GetAllProperties(PropertyType
                                          **ppPropertyVarType) = 0;
      //Abfrage einer Property
      //Parameter:
      //iPropID: ID der Property
```

```
      //pProp: Zeiger für Rückgabe
      virtual HRESULT STDMETHODCALLTYPE GetProperty(int iPropID,
                                                PropertyType *pProp) = 0;
      //Festlegen einer Property
      //Parameter:
      //Prop: die Property
      virtual HRESULT STDMETHODCALLTYPE SetProperty(PropertyType Prop) = 0;

      //Festlegen von Parametern zur Abstimmung unter allen Filtern
      //Parameter:
      //iParamID: Parameter-ID
      //dValue: Wert des Parameters
      virtual HRESULT STDMETHODCALLTYPE SetParam(int iParamID, double
                                                          dValue) = 0;

      //Zuweisen des Audiofokus'
      //Parameter:
      //bShowData: Flag
      virtual HRESULT STDMETHODCALLTYPE SetShowAudioData(bool bShowData)
                                                                  = 0;

};
```

D.4 Strukturen

D.4.1 AudioBuffer

Wrapper-Struktur für Audiodaten, dient dem Datenaustausch zwischen Host und Filtern

```
struct AudioBuffer
{
      PBYTE pData;                  //Zeiger auf Datenblock
      REFERENCE_TIME rtTimestamp;   //Zeitstempel
      int iLen;                     //Länge des Blocks in Byte
      int iSampleRate;              //Abtastrate
      int iBytesPerSample;          //Anzahl Bytes je Sample (je Kanal)
      int iChannels;                //Anzahl Kanäle je Sample
};
```

D.4.2 EventSample

Datenstruktur für die Verarbeitung von Events, auch von Filtern genutzt

```
typedef struct tagEVENTSAMPLE
{
      LONGLONG rtTimestamp;         //Zeitstempel
      char sDescriptor[512];        //Eventbeschreibung
      long lScore;                  //Score (Ranking)
      long lShift;                  //zeitl. Offset gegenüber Datenbank

      char sGenerator[256];         //Name des erzeugenden Filters
      int iSampleRate;              //Abtastrate
      int iBytesPerSample;          //Bytes je Sample (je Kanal)
      int iChannels;                //Kanäle je Sample

      long index;                   //fortlaufende Nummer
      char sVisualizer[1024];       // Name des visualisierenden Filters

      char sFutureUse[512];         //512 Byte, reserviert für Erweiterungen
} EventSample;
```

D.4.3 EventSampleWrapper

Wrapper-Struktur für die Verwendung der EventSample-Struktur in der Visualisierung

```
struct EventSampleWrapper
{
      EventSample es;              //Event-Objekt
      bool bDrawn;                 //Flag: Event bereits angezeigt?
};
```

D.4.4 EventTrack

Repräsentiert eine Sub-Bahn in CEventView

```
struct EventTrack
{
      char sDescription[1024];     //Eventbeschreibung
      int iLastIndex;              //Track-Index
};
```

D.4.5 FavouriteFilters

Struktur zur Verwaltung von Favoriten im CavailFilterList-Dialog

```
struct FavoriteFilters
{
      CString sName;               //Filterbezeichnung
      CLSID clsID;                 //Filter-CLSID
};
```

D.4.6 FilterDesc

Repräsentiert ein Event-visualisierendes Filter in CEventView

```
struct FilterDesc
{
      CString sName;                     //Filtername
      int color;                         //Farbe (RGB)
      int index;                         //aktueller Index
      POSITION pos;                      //Speicher-Index in CList
      EventSample esPrevEvent;           //voriges Event dieses Typs
      REFERENCE_TIME rtEventBlockBegin;  //Zeitstempel des Blockbeginns
      EventTrack tracks[16];             //Sub-Bahnen, max.16 Stück
      int iTrackCount;                   //deren tatsächliche Anzahl
};
```

D.4.7 EVENTINFO

Struktur zur Definition des Datentyps “Event” zur Inter-Filter-Kommunikation

```
typedef struct tagEVENTINFO
{
      int iVersion;                //Versionsnummer

} EVENTINFO;
```

D.4.8 PropertyType

Struktur zur Speicherung und Verarbeitung von Properties in Filtern

```
//Variablentyp-Konstanten
const PROPTYPE_INT = 1;
const PROPTYPE_STRING = 2;
const PROPTYPE_FLOAT = 3;
const PROPTYPE_FILE = 4;

typedef struct tagPROPERTYTYPE
{
      char sName[64];         //anzuzeigender Name
      int iPropID;            //ID
      int iPropType;          //Variablentyp
      void* pValue;           //Zeiger auf die Variable

      //Wertehalter, für Dialogkommunikation
      char sValue[255];       //Stringwert
      int iValue;             //Integerwert
      double fValue;          //Floatingwert

} PropertyType;
```

D.4.9 ViewBuffer

Repräsentiert in der Audio-Visualisierung einen Bereich, der ein Pixel breit ist

```
struct ViewBuffer
{
      int iValueMax;    //Maximalwert des repräsentierten Zeitintervalls
      int iValueMin;    //Minimalwert des repräsentierten Zeitintervalls
      REFERENCE_TIME rtTimestamp;   //Zeitstempel
};
```

Anhang E

Technische Dokumentation des Filter-SDK

Da das SDK wegen des Basierens auf Präprozessoranweisungen weniger aus einer komplexen Anzahl von Klassen, sondern vielmehr einer vernetzten Struktur ineinander eingebundener Dateien besteht, soll nachfolgend zunächst eine Übersicht über diese Struktur gegeben werden. In darauffolgenden Abschnitten werden wichtige Klassen und Dateien erläutert, bevor abschließend die Pin-Klassen und die zentrale Klasse `CMonitoringBaseFilter` besprochen werden.

Die im Rahmen dieser Diplomarbeit implementierten Filter folgen alle dem durch das SDK vorgegebenen Weg und unterscheiden sich nur durch den Code der einzelnen Funktionen. Da sie sowohl in Kapitel 5.3 als auch im Quellcode hinreichend dokumentiert sind und im Hinblick auf das zu Beginn des Anhangs D erläuterte Konzept des Fokus auf Erweiterbarkeit hier nur wenig Sinn ergäben, befasst sich dieses Kapitel des Anhangs ausschließlich mit Details des SDK und soll so dem interessierten Benutzer die Möglichkeit zu Anwendung und Erweiterung des SDK bieten.

Dieses Kapitel ist unterteilt in vier Abschnitte: Zunächst wird in E.1 ein Überblick über die Vernetzung der im SDK enthaltenen Dateien gegeben. Es folgt eine Erläuterung der wichtigsten Elemente in E.2, bevor in E.3 und E.4 die zentralen Klassen der Pins und CMonitoringBaseFilter besprochen werden sollen.

E.1 Überblick über die Vernetzung der eingebundenen Dateien

Die Abbildung E.1 zeigt das Netz der Verbindungen der Headerdateien des Filter-SDK. Zusätzlich spielt die Stelle der jeweiligen *Include*-Anweisung, in diesem Fall äquivalent zum Zeitpunkt der Einbindung und der daraus resultierenden Kompilierung.

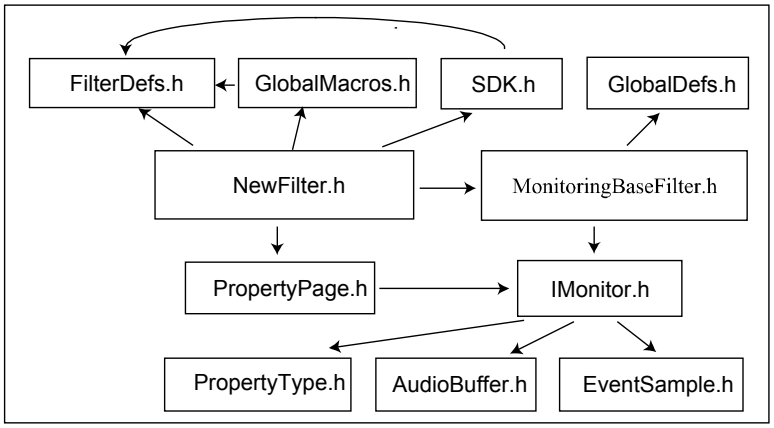


Abbildung E.1: Vernetzung der Headerdateien des Filter-SDK

E.2 Wichtige Klassen, Strukturen und Dateien

E.2.1 FilterDefs.h

Diese Datei enthält die zentralen Angaben zum Filter:

```
//Anzahl der verschiedenen Pintypen
#define PINS_AUDIO_IO 0
#define PINS_EVENT_IN 1
#define PINS_EVENT_OUT 1

//====================================================================
// *DER* GUID, hier die Filter-CLSID
const GUID IID_MonitoringEventIncluder =
{ 0x4c03cfaa, 0xb667, 0x43ee, 0x96, 0xaa, 0xa1, 0xea, 0x66, 0xfa, 0x6e, 0xed };

//====================================================================
//zentrale Definitionen des Filters für globale Makros:

//CLSID des Filters
#define FILTER_CLSID          IID_MonitoringEventIncluder
//Filter-Klasse, muss von CMonitoringBaseFilter erben
#define FILTER_CLASS          CmonitoringEventIncluder
//Filtername, ASCII
#define FILTER_NAME           "MonitoringEventIncluder"
//Filtername, Unicode
#define FILTER_NAME_WIDE      L"MonitoringEventIncluder"
//Filter-Konfigurationsdialog-CLSID
#define FILTERPROP_CLSID      CLSID_MonitoringBaseFilter_PropertyPage
//und -Klasse
#define FILTERPROP_CLASS      CMonitoringBaseFilterPropertyPage
```

E.2.2 FilterUtils

`namespace FilterUtils`
Hierbei handelt es sich nicht um eine Klasse, sondern um einen Namespace, der nur zwei Funktionen enthält:

`bool FileExists(char* sPath);`
Überprüft, ob am angegebenen Pfad eine Datei existiert. Falls ja, wird TRUE zurückgeliefert.

`bool BrowseFileName(char* sFile, HWND hWnd, bool bFileMustExist);`
Öffnet einen Dialog zur Dateiauswahl. Wird eine Datei ausgewählt, liefert die Funktion TRUE zurück und enthält den gewählten Pfad in `sFile`. Der Parameter `hWnd` wird für das Erzeugen des Dialogs benötigt und stellt ein Fensterhandle dar. Das Flag `bFileMustExist` schließlich bestimmt, ob die zu wählende Datei existieren muss, um auswählbar zu sein.

E.2.3 GlobalMacros.h

In dieser SDK-internen Datei werden Medientypen definiert und die zur Registrierung benötigten globalen Variablen deklariert und anhand der Angaben in `FilterDefs.h` gefüllt. Benutzer sollten diese Datei nur in Ausnahmefällen editieren!

E.2.4 IMonitor.h

In dieser Datei sind sowohl grundlegende Medientypen wie MEDIATYPE_Event als auch die elementaren IMonitoringMaster und –Slave-Schnittstellen und deren GUIDs definiert. Diese werden in Anhang D.3 erläutert. IMonitoringSlave wird von CMonitoringBaseFilter implementiert.

E.2.5 CMonitoringBaseFilterPropertyPage

`class CMonitoringBaseFilterPropertyPage : public CBasePropertyPage`
Diese Klasse stellt die im SDK standardmäßig verwendete Konfigurationsansicht dar und wird als solche in FilterDefs.h registriert. Basierend auf der DirectShow-Klasse CBasePropertyPage verwendet die Klasse den PropertyType-Datentyp zur Übergabe von Properties und überschreibt eine Reihe von Ereignisbehandlungsmethoden zum Umgang damit. Ein solches Vorgehen ist nötig, da es keine direkte Verbindung zwischen Filter und Konfigurationsseite gibt, sondern nur durch COM/DirectShow vermittelte Kommunikation. Prinzipiell werden – z.B. für dem Fall komplexer Filter – beliebig viele Konfigurationsseiten innerhalb des Dialogs unterstützt, hier wird jedoch nur eine Seite implementiert. Die Seite besteht aus acht Zeilen, die jeweils ein Label, ein Textfeld und einen Button beinhalten. In Abhängigkeit vorhandener Properties werden Zeilen mit Angaben, Namen gefüllt und teilweise oder komplett versteckt. Dadurch wird das Anzeigen dynamisch vorhandener Properties ermöglicht.
Durch Verwenden zweier Felder von Properties ist auch ein Zurücknehmen von Änderungen möglich:

```
PropertyType *pPropertyType;        //bisherige Werte
PropertyType *pPropertyTypeNewVal;  //neue Werte
int cPropertyCount;                 //Anzahl Properties

//Hilfsfunktion, die Properties Feld-weise kopiert
void CopyPropertyValue(PropertyType *to, PropertyType *from, int Count);
```

`CUnknown * WINAPI CreateInstance(LPUNKNOWN pUnk, HRESULT *pHr)`
Diese Factory-Methode erzeugt eine `CMonitoringBaseFilterPropertyPage`-Instanz und liefert diese zurück.

`HRESULT OnConnect(IUnknown *pUnk)`
Dieser Handler wird beim Verbinden der Applikation mit dem Konfigurationsdialog aktiviert und ermittelt durch Verwenden des `IMonitoringSlave`-Interfaces die im vorliegenden Filter veröffentlichten Properties.

`HRESULT OnActivate(void)`
Dieser Handler wird aufgerufen, sobald die Seite aktiviert wird. Hier findet das Füllen und Verstecken der Steuerelemente der acht Zeilen statt.

`HRESULT OnApplyChanges(void)`
Wird aufgerufen, wenn der Benutzer *Übernehmen* wählt. Hier werden die neuen Feldwerte übernommen.

`BOOL OnReceiveMessage(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)`
Diese Methode wird bei Veränderung von Steuerelementen der Konfigurationsseite aufgerufen und enthält Code zur Behandlung aller möglichen Elemente. Dabei werden unter Verwendung der `SetNewValue()`-Funktion die neuen Werte vermerkt.

`HRESULT OnDeactivate(void)`
Diese Methode wird sowohl beim Wechsel zu einer anderen (in diesem Fall nicht vorhandenen) Seite als auch beim Beenden des Dialogs aufgerufen. Falls der Benutzer neue Werte eingegeben hat, werden diese gespeichert.

`HRESULT OnDisconnect(void)`
Aufruf beim Verlassen des Dialogs. Auch hier werden die Änderungen übernommen.

E.2.6 PropertyType

Diese Datenstruktur wird in Anhang D.4.8 erläutert.

E.2.7 SDK.h

In dieser zentralen SDK-Datei befinden sich Präprozessorbefehle, die – basierend auf den Angaben aus FilterDefs.h – wichtige Methoden in den Code der Filterklasse einfügen, z.B. die Factory-Methode zum Erzeugen einer Instanz. Darüber hinaus werden Makros definiert, die Pin-Felder mit Pin-Instanzen füllen. Auf diese Weise wird ein ein Arbeiten in der zu erstellenden Filterklasse ermöglicht, als ob der Benutzer selbst diese Angaben manuell gemacht hätte.

E.2.8 DXAudioBuffer

Diese Stuktur wird in CMonitoringBaseFilter intern zum Datentransport verwendet.

```
struct DXAudioBuffer
{
      long cSamp;             // Anzahl Samples im Buffer
      IMediaSample* pms;      // das DirectX-IMediaSample für diesen Buffer
};
```

E.2.9 GlobalDefs.h

Diese Headerdatei beinhaltet die CLSIDs von `CMonitoringBaseFilter` und `CMonitoringBaseFilter_PropertyPage`.

E.2.10 EventSample.h

Diese Headerdatei enthält die in Anhang D.4.2 abgebildete `EventSample`-Datenstruktur.

E.3 Die Pin-Klassen

Im Folgenden sollen die wichtigen Methoden der Pin-Klassen erläutert werden. Dabei werden wichtige Operationen durch ein serialisierendes Streaming-Lock vom Typ `CCritSec` geschützt, das sich mittels Initialisierung durch die `CAutoLock`-Klasse automatisch selbst wieder öffnet.

E.3.1 CMonitoringAudioInputPin

```
class CMonitoringAudioInputPin : public CBaseInputPin
```
Diese Klasse repräsentiert einen Eingangs-Audiopin.

```
HRESULT CheckMediaType( const CMediaType* pmt )
```
In dieser Methode findet ein Teil der Kontrolle über Datenformate statt, die der Pin und somit das Filter für eingehende Daten akzeptieren. Wie in Abschnitt 5.2.1 erläutert findet hier nur ein prinzipielles Ablehnen von Nicht-Audioformaten des übergebenen Parameters statt; das eigentliche Überprüfen wird an das Filter deligiert.

```
HRESULT Receive( IMediaSample* pms )
```
Hier werden eingehende `IMediaSample`-Datenblöcke in DXAudioBuffer-Objekte verpackt an das Filter zur Weiterverarbeitung in `PreProcessAudio()` weitergereicht und abschließend per `DeliverOutputBuffer()`-Funktion des Filters zu nachgeschalteten Filtern transportiert. Im Fall einer Stream-Unterbrechung wird dies hier zusätzlich mit einem Zeitstempel festgehalten.

```
HRESULT EndOfStream()
```
Am Ende des Datenstroms aufgerufen sorgt diese Methode dafür, dass sämtliche noch vorhandene Datenblöcke – wie in `Receive()` – verarbeitet und weitertransportiert werden.

E.3.2 CMonitoringAudioOutputPin

```
class CMonitoringAudioOutputPin : public CBaseOutputPin
```
Diese Klasse repräsentiert einen Ausgangs-Audiopin.

```
HRESULT CheckMediaType( const CMediaType* pmt )
```
Der Vorgang ist nahezu identisch mit dem in `CMonitoringAudioInputPin`, hier wird jedoch zusätzlich noch darauf geachtet, dass der Eingangs-Audiopin überhaupt verbunden ist.

```
HRESULT DecideBufferSize( IMemAllocator* pAllocator,
                          ALLOCATOR_PROPERTIES* pProp )
```
Um den Transport von Ausgangs-Pin zu Eingangs-Pin zweier Filter zu ermöglichen, verwenden diese im Push-Modus ein sogenanntes `IMemAllocator`-Objekt, das hier als Parameter übergeben wird. Dessen ebenfalls übergebene Eigenschaften bestimmen die Details des Transportes und werden an dieser Stelle abgestimmt.

```
STDMETHODIMP EnumMediaTypes( IEnumMediaTypes** ppEnum )
```
Diese Methode liefert per übergebenem Parameter eine Liste aller präferierten Medientypen zurück, in dem versucht wird, diese Funktion auf denjenigen Pin anzuwenden, der mit dem Eingangs-AudioPin des Filters verbunden ist. Die Anfrage wird also nach Möglichkeit *stromaufwärts* weitergeleitet.

`STDMETHODIMP NonDelegatingQueryInterface( REFIID riid, void**ppv )`
Die von dieser Klasse an vorgeschaltete Filter weitergeleiteten `IMediaPosition`- bzw. `IMediaSeeking`-Anfragen werden hier veröffentlicht.

E.3.3 CMonitoringEventInputPin

`class CMonitoringEventInputPin : public CBaseInputPin`
Diese Klasse repräsentiert einen Eingangs-Eventpin.

`HRESULT CheckMediaType( const CMediaType* pmt )`
Analog zum Vorgehen in den Audiopins werden hier nur Nicht-Eventformate abgelehnt, bevor die Anfrage an das Filter weitergereicht wird.

`HRESULT Receive( IMediaSample* pms )`
Diese Methode reicht das unbearbeitete Eventobjekt, welches noch in ein von DirectShow zum Transport verwendetes `IMediaSample`-Objekt verpackt ist, an die Filter-Methode `PreProcessEvent()` weiter.

E.3.4 CMonitoringEventOutputPin

`class CMonitoringEventOutputPin : public CBaseOutputPin`
Diese Klasse repäsentiert einen Ausgangs-Eventpin.

`STDMETHODIMP NonDelegatingQueryInterface( REFIID riid, void**ppv )`
Die von dieser Klasse an vorgeschaltete Filter weitergeleiteten `IMediaPosition`- bzw. `IMediaSeeking`-Anfragen werden hier veröffentlicht.

`HRESULT CheckMediaType( const CMediaType* pmt )`
Der Vorgang ist inhaltlich identisch mit dem in `CMonitoringEventInputPin.`

```
HRESULT DecideBufferSize( IMemAllocator* pAllocator,
                                          ALLOCATOR_PROPERTIES* pProp )
```
Genau wie im Fall der Audiopins werden auch hier Eigenschaften des `IMemAllocator`-Objektes mit verbundenen Pins abgestimmt. Dazu wird zunächst versucht, die Eigenschaften eines Eingangs-Eventpins zu verwenden. Ist ein solcher nicht vorhanden, werden die wichtigsten Eigenschaft, Anzahl und Größe der verwendeten Puffer, auf zwei Puffer der Größe eines `EventSample`-Objektes festgesetzt.

`STDMETHODIMP EnumMediaTypes( IEnumMediaTypes** ppEnum )`
Diese Methode liefert einen Enumerator zum Auflisten aller präferierten Medientypen zurück.

`HRESULT DeliverEvent( IMediaSample* pms )`
Diese Methode liefert einen übergebenen Event an einen verbundenen Pin durch direkte Verwendung von DirectShow-Funktionalität.

E.4 CMonitoringBaseFilter

```
class CMonitoringBaseFilter :
      public CBaseFilter,                 //DirectShow Filter
      public IspecifyPropertyPages,       //Property Pages
      public CPersistStream,              //Speichern interner Variablen
      public IMonitoringSlave,            //"unser" Interface
      public IMediaSeeking                //Position setzen
```

Diese Klasse beinhaltet Filter-spezifische Themen wie Datentransport, Format-Abgleich und andere. Ein neues, zu erstellendes Filter, das von dieser Klasse ableitet, kann dadurch auf einem festen Fundament aufbauen, das sämtliche Low-Level-Arbeit abdeckt und dem Entwickler die notwendige Verantwortung abnimmt. Dadurch kann dieser sich auf die wesentlich Bestandteile des neuen Filters konzentrieren und somit die Produktivität nachhaltig steigern. Alle im Rahmen dieser Diplomarbeit vorgestellten Filter leiten direkt von CMonitoringBaseFilter ab.

CMonitoringBaseFilter ist eine Klasse, die zwischen DirectShow-Funktionalität und GenMAD-Kommunikation vermittelt. Dazu bedient sie sich einer Reihe wichtiger Eigenschaften:

```
char sFiltername[255];        //GraphController-Name des Filters

PBYTE pShowAudioBuf;          //Datenpuffer für zu anzuzeigende Audiodaten
IMemAllocator* m_pAllocator;  //Allokator zur Übergabe der Daten
LONGLONG m_llSamplePosition;  //aktuelle Stream-Position
CCritSec m_Lock;              //Lock: Zugriff wird serialisiert

//die Pins, maximal je zwei Eventpins
CMonitoringAudioInputPin *pAudioInputPin;
CMonitoringAudioOutputPin *pAudioOutputPin;
CMonitoringEventInputPin *pEventInputPin[2];
CMonitoringEventOutputPin *pEventOutputPin[2];

IMonitoringMaster *pMonitoringMaster;    //Kommunication mit GenMAD

//veröffentlichte Properties
PropertyType *pPropertyType;
int cPropertyCount;

//Zeitstempel des aktuellen (evtl. noch offenen) Puffers
REFERENCE_TIME ref_s;
REFERENCE_TIME rtSeekOffset;

//eigenes puffer management:
int iBufferSize;              //Puffergrösse in Byte
int iDirectXBufferSize;       //aktuelle DirectShow-Puffergrösse
bool bUseSpecificBuffer;      //eigenen Puffer verwenden?
BYTE *pSpecificBuffer;        //der eigene Puffer
BYTE *pBuffer;                //Zeiger auf aktuelle Puffer-Schreibestelle
```

Nachfolgend werden die wesentlichen Methoden erläutert:

`void Init()`
In dieser Funktion, die unter anderem von den Konstruktoren aufgerufen wird, ist die zentrale Initialisierung aller Variablen enthalten.

```
virtual HRESULT CallSlave(int iCodeID, void* pParam, void** ppReturnValue);
virtual HRESULT SetIMonitoringMaster(IMonitoringMaster *pmm);
virtual int GetAllProperties(PropertyType **ppPropertyVarType);
virtual HRESULT GetProperty(int iPropID, PropertyType *pProp);
virtual HRESULT SetProperty(PropertyType Prop);
virtual HRESULT SetShowAudioData(bool bShowData);
virtual void FreeResources();
virtual int Ready(char *szErrMsg);
virtual void SetBufferSize(int iBytes);
virtual int GetBufferSize() {return iBufferSize;};
virtual HRESULT SetParam(int iParamID, double dValue);
virtual void SetFilterName(char* sNewName);
```

Diese Methoden sind die Implementation des `IMonitoringSlave`-Interfaces, welches im Anhang D.3.2 erläutert ist.

`CUnknown * WINAPI CreateInstance(LPUNKNOWN punk, HRESULT *phr)`
Factory-Methode zum Erzeugen des Filters.

`void SendNotify(long EventCode)`
Versenden von DirectShow-Ereignissen mit übergebenem Code.

`HRESULT SendEvent(EventSample* pes)`
Senden eines übergebenen Datensatzes vom Typ *Event* an alle nachgeschalteten Filter. Dazu wird der Event binär serialisiert und als Block an alle verbundenen Ausgangs-Eventpins versandt.

`HRESULT PreProcessEvent(IMediaSample *pSample, int iPinID)`
Hier findet der zu `SendEvent()` entgegengesetzte Prozess statt, in dem aus einem binären `IMediaSample`-Datenblock ein Event deserialisiert wird, das an `ProcessEvent()` weitergereicht wird. Der Parameter `iPinID` definiert dabei den Pin, der die Daten empfangen hat.

`HRESULT ProcessEvent(EventSample es, int iPinID)`
Von abgeleiteten Filterklassen zu implementierende Methode zur Behandlung eines übergebenen Events. Der Parameter `iPinID` definiert dabei den Pin, der die Daten empfangen hat.

`STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void **ppv)`
Bekanntgabe implementierter Schnittstellen: `IMonitoringSlave`, `ISpecifyPropertyPages`, `IPersistStream` und `IMediaSeeking`.

`HRESULT GetPeerSeeking(IMediaSeeking ** ppMS)`
Hilfsfunktion für Seeking-Funktionalität: Diese Methode versucht ein stromaufwärts implementiertes `IMediaSeeking`-Interface zurück zu liefern.

```
HRESULT GetSeekingLongLong(
   HRESULT (__stdcall IMediaSeeking::*pMethod)(LONGLONG*), LONGLONG *pll)
```

Eine zweite Hilfsfunktion für das Seeking: Diese liefert zu einem gegebenen IMediaSeeking die Adresse einer Funktionsimplementation der Schnittstelle und somit eine Sprungadresse für den Funktionsaufruf.

```
virtual GetCapabilities( DWORD *pCapabilities );
virtual CheckCapabilities( DWORD *pCapabilities );
virtual SetTimeFormat(const GUID *pFormat);
virtual GetTimeFormat(GUID *pFormat);
virtual IsUsingTimeFormat(const GUID *pFormat);
virtual IsFormatSupported( const GUID *pFormat);
virtual QueryPreferredFormat( GUID *pFormat);
virtual ConvertTimeFormat(LONGLONG *pTarget, const GUID *pTargetFormat,
```

```
                            LONGLONG Source, const GUID *pSourceFormat);
virtual SetPositions( LONGLONG *pCurrent, DWORD CurrentFlags,
                      LONGLONG *pStop, DWORD StopFlags );
virtual GetPositions( LONGLONG *pCurrent, LONGLONG *pStop );
virtual GetCurrentPosition( LONGLONG *pCurrent );
virtual GetStopPosition( LONGLONG *pStop );
virtual SetRate( double dRate);
virtual GetRate( double *pdRate);
virtual GetDuration( LONGLONG *pDuration);
virtual GetAvailable( LONGLONG *pEarliest, LONGLONG *pLatest );
virtual GetPreroll( LONGLONG *pllPreroll );
```

Diese Methoden stellen die Implementierung des IMediaSeeking-Interfaces dar. Allerdings leiten sämtliche Methoden die eingehenden Anfragen per `GetPeerSeeking()` oder `GetSeekingLongLong()` an stromaufwärts verbundene Filter weiter, falls möglich. Ansonsten wird ein Fehlercode zurück geliefert.

```
HRESULT PreProcessAudio(IMediaSample *pSample)
```

In dieser Methode findet die gesamte Vorverarbeitung eines übergebenen Audiodatenblocks statt: Dazu werden zunächst Block-Eigenschaften wie Größe, Zeitstempel und Adresse der eigentlichen Daten ermittelt. Falls das aktuelle Filter den Audiofokus zugewiesen hat, wird eine `AudioBuffer`-Objekt mit den Daten befüllt und an GenMAD übergeben. Im Anschluss findet, falls vorgegeben, die Anpassung auf eine benutzerdefinierte Puffergröße statt; dazu werden die Daten sukzessiv in einen anderen Puffer namens `pSpecificBuffer` kopiert und an `ProcessAudio()` übergeben. Falls am Ende der Routine Samples nicht verarbeitet wurden, bleiben diese in `pSpecificBuffer` bis zum nächsten Aufruf erhalten.

```
HRESULT ProcessAudio(PBYTE pb, int cb, LONGLONG timestamp)
```

Von abgeleiteten Filterklassen zu implementierende Methode zur Behandlung eines übergebenen Audioblocks `pb` der Länge `cb`. Der Parameter `timestamp` definiert dabei den Zeitstempel des Blocks.

```
HRESULT STDMETHODCALLTYPE GetPages(CAUUID *pPages)
```

Die Schnittstelle `ISpecifyPropertyPages` implementierende Methode zur Übergabe von Anzahl und GUID der verwendeten Konfigurationsseiten.

```
virtual HRESULT WriteToStream(IStream *pStream);
virtual HRESULT ReadFromStream(IStream *pStream);
virtual int SizeMax();
virtual GetClassID(CLSID *pClsid);
virtual DWORD GetSoftwareVersion(void);
```

Implementation des `IPersistStream`-Interfaces. Dazu werden Werte mittels eines `IStream`-Objektes serialisiert und zur Speicherung zurückgegeben bzw. aus einem `IStream`-Objekt deserialisiert und in die entsprechenden Variablen übertragen. Die anderen Methoden sind Hilfsfunktionen.

```
void STDMETHODCALLTYPE SetBufferSize(int iBytes)
```

Diese Methode wird zu Einstellen einer benutzerdefinierten Audio-Puffergröße `iBytes` aufgerufen und ändert diese je nach Abspielstatus entweder direkt oder setzt wegen der Verwendung der zu ändernden Puffer ein entsprechendes Flag, welches in `PreProcessAudio()` als Indikator zur verzögerten Änderung verstanden wird.

```
void SetBufferSizeExec(int iBytes)
```

Führt die Änderung der neuen Puffergröße `iBytes` durch.

```
bool InTime(REFERENCE_TIME rfBlockTime)
```
Diese Methode ermöglicht durch Überschreiben das Ausfiltern von Audioblöcken, deren Zeitstempel eine gewissene Latenz überschreitet, z.B. im Fall nicht echtzeitfähiger Algorithmen. Dazu wird die Methode mit dem zu untersuchenden Zeitstempel von `PreProcessAudio()` aufgerufen und zeigt das nötige Ausfiltern durch Zurückliefern von FALSE an.

```
virtual STDMETHODIMP Stop();
virtual STDMETHODIMP Pause();
virtual STDMETHODIMP Run(REFERENCE_TIME tStart);
```
Diese drei grundlegenden Funktionen rufen jeweils zunächst die entsprechende Methode der Basisklasse auf und behandeln anschließend ein eventuell auftretendes Ende des Datenstromes.

```
HRESULT CheckMediaType( PIN_DIRECTION pinDir, const CMediaType* pmt )
HRESULT CheckTransform( const CMediaType* pmtIn, const CMediaType* pmtOut )
HRESULT SetMediaType( PIN_DIRECTION pindir, const CMediaType* pmt )
```
Unter Verwendung von Parametern zur Flussrichtung des betreffenden Pins und der zu untersuchenden Medienformatee bestimmen diese drei Methoden, welche Datentypen direkt bzw. nach Transformation akzeptiert werden können und erhalten eine Benachrichtigung über das vereinbarte, verbindliche Format.

```
HRESULT GetOutputBuffer( CMonitoringAudioOutputPin* pOutputPin,
      DXAudioBuffer* pbufOut, REFERENCE_TIME* prtStart, BOOL bSyncPoint,
      BOOL bDiscontinuity, BOOL bPreroll )
HRESULT DeliverOutputBuffer( CMonitoringAudioOutputPin* pOutputPin,
      DXAudioBuffer* pbufOut, HRESULT hrProcess, BOOL bCleanup )
```
Diese beiden überschriebenen Methoden erzeugen unter Verwendung der Parameter einen neuen Audiopuffer und liefern diesen über den Ausgangs-Audiopin aus.

```
HRESULT EndOfStream()
```
Diese Methode garantiert im Fall einer benutzerdefinierten Puffergröße die Abarbeitung noch offener Samples per `ProcessAudio()`, falls das Ende des Datenstromes signalisiert wird.

Anhang F

Bedienung des Klassifikationsfilters

Das MonitoringClassifier-Filter operiert in sieben verschiedenen Modi:

Modus 0: Erkennungsmodus, arbeitet auf im k-d-Baum abgelegten Trainingsdaten. Klassifikationen werden als Events an nachgeschaltete Filter versandt.

Modus 1: Aus Trainingsdaten den k-d-Baum erstellen. Dies kann je nach Datenmenge einige Zeit in Anspruch nehmen, in der das Filter nicht reagiert.

Modus 2: Laden des Baumes aus einer Datei

Modus 3: Speichern des Baumes in eine Datei

Modus 4: Trainingsdaten für Klasse Musik

Modus 5: Trainingsdaten für Klasse Sprache

Modus 6: Reset aller Baumdaten

Um Audiosignale zu klassifizieren, muss entweder ein bereits trainierter Baum aus den in einer Datei abgelegten Daten geladen (Modus 2) oder selbst eine Menge von Trainingsdaten zur Konstruktion verwendet werden (Modi 4/5 zum Training, danach Modus 1 für die Erzeugung des Baumes). Wird ein Klassifikatorfilter durch Laden eines Projektes aufgerufen, so wird versucht, die entsprechenden Baumdaten automatisch zu laden.

Entsprechend umfasst der Konfigurationsdialog folgende Parameter:

Index-Dateiname: Datei für das Laden und Speichern des Baumes

k: Parameter des k-NN-Klassifikators

Modus: aktueller Operationsmodus

Stille_Schwellwert: Schwellwert der Stille-Erkennungsstufe

Anhang G

Inhalt der beiliegenden CD

Auf der beiliegenden CD befindet sich die folgende Verzeichnisstruktur:

```
\AudentifyFilter
\BaseFilter
\Classifier
\DirectX
\EventExcluder
\EventIncluder
\EventMerger
\EventNullRenderer
\EventSplitter
\EventVisualizer
\Interfaces
\Monitor
\SDK
\WaveDest
\XmlReader
\XmlWriter
GenMAD.exe
MonitoringAudentifyFilter.ax
MonitoringClassifier.ax
MonitoringEventExcluder.ax
MonitoringEventIncluder.ax
MonitoringEventMerger.ax
MonitoringEventNullRenderer.ax
MonitoringEventSplitter.ax
MonitoringEventVisualizer.ax
MonitoringXMLReader.ax
MonitoringXMLWriter.ax
pcmIndexDLL.dll
wavdest.ax
```

Dabei handelt es sich um die Verzeichnisse mit den Quelltexten und den Projektdateien der einzelnen Filter und von GenMAD, welches sich im Verzeichnis `Monitor` befindet. Hinzu kommen Verzeichnisse mit gemeinsamen Elementen. Im Wurzelverzeichnis befinden sich die kompilierten Versionen aller Komponenten, die nach der Registrierung (siehe Anhang B) direkt lauffähig sind.

Literaturverzeichnis

[1] P. Ackermann, "Developing Object-Oriented Multimedia Software - Based on MET++ Application Framework", dpunkt Verlag, Heidelberg, 1996.

[2] E. Allamanche, J. Herre, O. Helmuth, B. Fröba, T. Kasten and M. Cremer, "Content-based identication of audio material using mpeg-7 low level description", *in Proc.of the Int. Symp. of Music Information Retrieval (ISMIR)*, Indiana, USA, Oct.2002.

[3] M. Alonso, B. David and G. Richard, "Tempo and beat estimation of music signals", *Proc. of ISMIR 2004*, Barcelona, Spain, Oct. 2004.

[4] ALSA, Advanced Linux Sound Architecture Projekt Homepage
`http://www.alsa-project.org`

[5] Apple iTunes
`http://www.itunes.de`

[6] Apple Logic Pro
`http://www.apple.com/logic`

[7] Apple Quicktime
`http://developer.apple.com/quicktime`

[8] J.-J. Aucouturier and M. Sandler, "Segmentation of Musical Signals Using Hidden Markov Models", *Proceedings of the Audio Engineering Society 110th Convention*, May 2001.

[9] Audioble Magic
`http://www.audiblemagic.com`

[10] AudibleMagic Press, "Indies Join Major Labels In Fingerprinting Songs For Copyright Protection With Audible Magic's RepliCheck Service", 25.10.2004.
`http://www.clango.com/news&press/press_20041025.html`

[11] R. Baeza-Yates and B. Ribiero-Neto, „Modern Information Retrieval", Addison Wesley, 1999.

[12] B. Bailey, J. A. Konstan, R. Cooley and M. Dejong, "Nsync – A toolkit for building interactive multimedia presentations", *Proceedings of the 6th ACM International Conference on Multimedia'98*, ACM Press, 257-266, 1998.

[13] R.Bardeli, „Effiziente Algorithmen zur deformationstoleranten Suche in Audiodaten", Diplomarbeit, Universität Bonn, 2003.

[14] E. Batlle, J. Masip, E. Guaus, "Amadeus: A Scalable HMM-based Audio Information Retrieval System", *Proceedings of First International Symposium on Control, Communications and Signal Processing*, Hammamet, Tunisia, 2004.

[15] Bay TSP – Tracking, Security & Protection
`http://www.baytsp.com`

[16] R. Bellman, "Adaptive Control Processes: A Guided Tour", Princeton University Press, 1961.

[17] C. M. Bishop, "Neural Networks for Pattern Recognition", Oxford University Press, Oxford, 1995.

[18] G. Booch, "Object Oriented Design with Applications", Benjamin/Cummings, Redwood City, 2. Auflage, 1994.

[19] Brand Detector
`http://www.detect-tv.com`

[20] C. J. C. Burges, J. C. Platt, S. Jana, "Extracting Noise-Robust Features from Audio Data", Microsoft Res., Redmond, WA; ICASSP, p. 1021-1024, 2002.

[21] P. Cano, E. Batlle, T. Kalker, and J. Haitsma, "Review of Audio Fingerprinting Algorithms", *in MMSP*, 2002.

[22] P. Cano, E. Batlle, H. Mayer, and H. Neuschmied, "Robust sound modeling for song detection in broadcast audio", *in Proc. AES 112th Int. Conv.*, Munich, Germany, May 2002.

[23] P. Cano, M. Koppenberger, S. Le Groux, J. Ricard, P. Herrera, N. Wack, "Nearest-neighbor generic sound classification with a wordnet-based taxonomy", *Proceedings of AES 116th Convention*, Berlin, Germany, 2004.

[24] Cedega
`http://www.transgaming.com`

[25] E. Chavez, G. Navarro, R. A. Baeza-Yates and J. L. Marroquin, "Searching in metric spaces", *ACM Computing Surveys*, vol.33, no.3, pp.273-321, 2001.

[26] Cycling 74 Max/MSP – a graphical environment for music, audio, and multimedia.
`http://www.cycling74.com/products/maxmsp.html`

[27] M. Davy and S. G. Godsill, „Audio information retrieval: a bibliographical study", *Technical Report CUED/F-INFENG/TR.429*, Cambridge University Engineering Department, 2002.

[28] L. deCarmo, "A new architecture for multimedia", *PC Magazine*, 6, 1998.

[29] D. Dingeldein, "Modeling multimedia objects with MME", *Proceedings of the EUROGRAPHICS Workshop on Object-Oriented Graphics (EOOG'94)*, 1994.

[30] DirectX Files
`http://www.directxfiles.com`

[31] C. Djeraba, H. Saadane, "Automatic Discrimination in Audio Documents", *IRIN*, Nantes University, o. J.

[32] P. J. O. Doets and R. L. Lagendijk, "Theoretical Modeling Of A Robust Audio Fingerprinting System", *Fourth IEEE Benelux Signal Processing Symposium*, 2004.

[33] M. Duarte and Y. H. Hu, "Vehicle classification in distributed sensor networks", *Journal of Parallel and Distributed Computing*, 2004.

[34] K. El-Maleh, M. Klein, G. Petrucci, and P. Kabal, "Speech music discrimination for multimedia applications", *In ICASSP*, vol. IV, pp. 2445-2448, 2000.

[35] G. Engels, S. Sauer, „Object-oriented modeling of multimedia applications", *In S.-K. Chang (ed.), Handbook of multimedia applications*, pp.21-53, World Scientific, 2002.

[36] M. E. Fayad and D. C. Schmidt, "Special issue on object oriented application frameworks", *Comm. of the ACM*, 40, October 1997.

[37] R. Ferber, „Data Mining und Information Retrieval", Kap. 1.3.7, dpunkt Verlag, 2003.

[38] J. Foote, „An overview of audio information retrieval", *Multimedia Systems*, vol.7 no.1, p.2-10, Jan. 1999.

[39] J. H. Friedman, J. L. Bentley and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time", *ACM Trans. Math. Software 3*, no. 3, 209-226, 1977.

[40] F. Fünfstück, R. Liskowsky, K. Meißner, „Softwarewerkzeuge zur Entwicklung multimedialer Anwendungen. Eine Übersicht", *Informatik Spektrum*, 23:1, 11-25, 2000.

[41] J. Garcia, V. Tarasov, E. Batlle, E. Guaus, J. Masip, "Industrial audio fingerprinting distributed system with CORBA and Web Services", *Proceedings of Fifth International Conference on Music Information Retrieval*, Barcelona, 2004.

[42] D. Gerhard, "Audio Signal Classification: History and Current Techniques", *Technical Report TR-CS 2003-7*, University of Regina Department of Computer Science, November 2003.

[43] Gesellschaft für deutsche Sprache (GfdS) – Wörter des Jahres
`http://www.gfds.de/woerter.html`

[44] GStreamer Projekt Homepage
`http://gstreamer.freedesktop.org`

[45] J. Haitsma and T. Kalker, "A highly robust audio fingerprinting system", *International Symposium on Musical Information Retrieval (ISMIR)*, 2002.

[46] Helix Player
`https://helixcommunity.org`

[47] P. Herrera-Boyer, G. Peeters, and S. Dubnov, "Automatic classification of musical instrument sounds", *Mosart Deliverabel D22, Evaluation report of Timbre modeling*, 2002.

[48] T. Hoeren, „Einführung in das Multimedia-Recht“, *Symposium „Internet zwischen Kunst und Kommerz“*, Wiesbaden, 1996.

[49] D. Hubbard, "A simple STL based XML parser"
`http://www.codeproject.com/cpp/stlxmlparser.asp`

[50] O. Izmirli, “Using a Spectral Flatness Based Feature for Audio Segmentation and Retrieval”, *Proceedings of the International Symposium on Music Information Retrieval (ISMIR2000)*, Plymouth, Massachusetts, USA, October 2000.

[51] JACK Projekt Homepage
`http://jackit.sourceforge.net`

[52] A. K. Jain, R. P. W. Duin and J. Mao, “Statistical pattern recognition: a review”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 1, pp. 4-37, 2000.

[53] Java Media Framework
`http://java.sun.com/products/java-media/jmf`

[54] R. E. Johnson and B. Foote, “Designing Reusable Classes”, *Journal of Object-Oriented Programming (JOOP)*, June/July 1988.

[55] E. R. Kandel, J. H. Schwartz, T. M. Jessell, “Principles of Neural Science”, 4th Edition, McGrawHill, 2000.

[56] D. Kimber and L. Wilcox, “Acoustic Segmentation for Audio Browsers”, *Proc. Interface Conference*, Sydney, Australia, July, 1996.

[57] F. Kurth, „Beiträge zum effizienten Multimediaretrieval“, Habilitationsschrift, Universität Bonn, 2004.

[58] F. Kurth, R. Scherzer, ”Robust Real-Time-Identification of PCM Audio Sources”, *In: Proc. 114th AES Convention*, Amsterdam, NL, 2003.

[59] A. T. K. Leong, “A Music Identification System Based on Audio Content Similarity”, work for the degree for Bachelor of Engineering, Univ. of Queensland, Oct. 2003.

[60] Linux New Media Award 2004
`http://www.linuxnewmedia.de/Award_2004/award2004`

[61] Linux Simple DirectMedia Layer Projekt Homepage
`http://www.libsdl.org`

[62] Z. Liu, Q. Huang, “Classification of Audio Events for Broadcast News”, *IEEE Workshop on Multimedia Signal Processing*, Los Angeles, 1998.

[63] B. Logan, “Mel Frequency Cepstral Coefficients for music modeling”, *International Symposium on Music Information Retrieval (ISMIR)*, 2000.

[64] MARSYAS – a software framework for computer audition.
`http://opihi.cs.uvic.ca/marsyas`

[65] M.F. McKinney and J. Breebaart, "Features for Audio and Music Classification", *in 4th International Conference on Music Information*, 2003.

[66] Media Transfer Protocol Specification, MTP
`http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwmt/html/mtp_spec.asp`

[67] Microsoft DirectX
`http://msdn.microsoft.com/directx`

[68] MIDI-Spezifikation
`http://www.midi.org/about-midi/specinfo.shtml`

[69] A. Moore, "K-means and Hierarchical Clustering", Carnegie Mellon Univ., 2001.

[70] MPEG-7 Overview (version 9), 2003.
`http://www.chiariglione.org/MPEG/standards/mpeg-7/mpeg-7.htm`

[71] MPEG-21 Overview 5, Oktober 2002.
`http://www.chiariglione.org/MPEG/standards/mpeg-21/mpeg-21.htm`

[72] MSN Music
`http://beta.music.msn.com`

[73] Music Reporter
`http://www.musicreporter.net`

[74] MusicBrainz
`http://www.musicbrainz.org`

[75] MusicMatch Jukebox
`http://www.musicmatch.com`

[76] MusicTrace, "Broadcast Monitoring and Internet Tracing for Music and Advertising"
`http://www.musictrace.de`

[77] Napster
`http://www.napster.com`

[78] Native Instruments Reaktor
`http://www.nativeinstruments.de`

[79] Nielsen Broadcast Data System
`http://www.bdsonline.com`

[80] A. Y. Nooralahiyan, L. Lopez, D. McKewon and M. Ahmadi, "Time-delay neural network for audio monitoring of road traffic and vehicle classification", *Proc. SPIE Vol. 2902*, pp. 191-198, 1997.

[81] A. V. Oppenheim, R. W. Schafer and J. R. Buck, "Discrete-Time Signal Processing", 2nd edition, Prentice-Hall, Inc., 1999.

[82] C. Panagiotakis and G. Tziritas, "A Speech/Music Discriminator Based on RMS and Zero-Crossings", *IEEE Transactions on Multimedia*, 2003.

[83] K. Pearson, "On lines and planes of closest fit to systems of points in space", *The London, Edinburgh and Dublin Philosophical Magazine and Journal of Science*, 1901.

[84] V. Peltonen, "Computational auditory scene recognition", M.Sc. thesis, Tampere University of Technology, Finland, 2001.

[85] Philips - Gracenote Joint Venture
`http://www.research.philips.com/initiatives/contentid`

[86] Real Networks
`http://www.real.com`

[87] Relatable
`http://www.relatable.com`

[88] RFC 1889, "RTP, real time transport protocol & RTCP, RTP control protocol"

[89] RFC 2326, "RTSP, Real Time Streaming Protocol"

[90] RIAA-IFPI, "Request for information on audio fingerprinting technologies", 2001.
`http://www.ifpi.org/site-content/press/20010615.html`

[91] A. Ribbrock, F. Kurth, "A Full-Text Retrieval Approach to Content-Based Audio Identification", *International Workshop on Multimedia Signal Processing,* St. Thomas, US Virgin Islands, December 2002.

[92] E. Scheirer, "Tempo and Beat Analysis of Acoustic Musical Signals", *in J. Acoust. Soc. Am. 103(1)*, pp 588-601, Jan 1998.

[93] E. Scheirer and M. Slaney, "Construction and evaluation of a robust multifeature speech music discriminator", *In ICASSP*, 1997.

[94] C. E. Shannon, "Communication in the presence of noise", *Proc. Institute of Radio Engineers*, vol. 37, no.1, pp. 10-21, Jan. 1949.

[95] J. O. Smith and J. S. Abel, "The Bark and ERB bilinear transforms", *IEEE Trans. Speech and Audio Processing*, vol. 76, pp. 697-708, Nov. 1999.

[96] J. O. Smith and P. Gossett, "A flexible sampling-rate conversion method", *in Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, San Diego*, vol. 2, (New York), pp. 19.4.1-19.4.2, March 1984.

[97] B. C. Smith, L. A. Rowe, J. A. Konstan and K. D. Patel, "The Berkeley Continuous Media Toolkit", *Proceedings of the 4th ACM International Conference on Multimedia'96*, ACM Press, pp. 451-452, 1996.

[98] Snocap
http://www.snocap.com

[99] Spiegel Online, "Rot-Grün setzt sich für Radioquote ein", 15.12.2004
http://www.spiegel.de/kultur/gesellschaft/0,1518,333030,00.html

[100] R. Steinmetz, „Multimedia-Technologie: Grundlagen, Komponenten und Systeme“, Springer Verlag 1999.

[101] N. Stiel, “Multimedia: the new frontier”, Encyclopaedia Universalis, pp.144-149, 1995.

[102] C. Sun, S. G. Ritchie and S. Oh, “Inductive Classifying Artificial Network for Vehicle Type Categorization”, *Computer-Aided Civil and Infrastructure Engineering 18 (3),* pp. 161-172, 2003.

[103] Taligent, “Building Object-Oriented Frameworks, A Taligent White Paper”, *Technical report*, Taligent Inc., 1994.

[104] Technology Review, "LABOR: Forschung und Entwicklung: Der Sinn der Musik".
http://www.heise.de/tr/artikel/44683/0

[105] G. Tzanetakis, “Manipulation, Analysis and Retrieval Systems for Audio Signals”, *Princeton Computer Science Technical Report TR-651-02* , June 2002.

[106] G. Tzanetakis and P. Cook, “MARSYAS: A Framework for Audio Analysis”, *Organized Sound*, Cambridge University Press 4(3), 2000.

[107] G. Tzanetakis and P. Cook, “Multi-Feature Audio Segmentation for Browsing and Annotation”, *In Proc. IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, New Paltz, NY, 1999.

[108] G. Tzanetakis, G. Essl and P. Cook, “Automatic Musical Genre Classification Of Audio Signals”, *presented at International Symposium on Music Information Retrieval*, Bloomington, Indiana, USA, 2001.

[109] G. Tzanetakis, J. Gao and P. Steenkiste, “A scalable peer-to-peer system for music content and information retrieval”, *Computer Music Journal (Special Issue on Music Information Retrieval)*, 2004.

[110] M. Vacher, D. Istrate, L. Besacier, J. F. Serignat and E. Castelli, “Sound Detection and Classification for Medical Telesurvey”, *Proc. Of the 2nd Intern. Conf. Biomedical Engineering*, Innsbruck, Austria, 2004.

[111] C. J. Van Rijsbergen, “Information Retrieval”, Butterworths, Boston, 1980.

[112] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm", *IEEE Transactions on Information Theory*, vol. 13, no2, pp.260-269, 2001.

[113] Vorlesungsskript Medientechnik, Universität Hamburg, WS 2001

[114] A. Wang. "An Industrial Strength Audio Search Algorithm". *Proc.of the Int. Symp. of Music Information Retrieval 2003*, Baltimore, 2003.

[115] H. Wessels, "Audio Information Retrieval: Überblick", Proseminar "Robuste Signalidentifikation", Universität Bonn, 2003/04.

[116] Wikipedia, Discrete cosine transform
`http://en.wikipedia.org/wiki/Discrete_cosine_transform`

[117] E. Wold, T. Blum, D. Keislar and J. Wheaton, "Content-based classification, search and retrieval of audio", *IEEE Multimedia Mag.*, vol. 3, pp. 27-36, July 1996.

[118] H. J. Wolfson and I. Rigoutsos, "Geometric hashing: An overview", *IEEE Computational Science and Engineering*, pp. 10-21, October-December 1997.

[119] W3C, "Semantic Web" Homepage
`http://www.w3.org/2001/sw`

[120] Yacast
`http://www.yacast.com`

[121] A. Zell, „Simulation neuronaler Netze", Addison-Wesley Verlag, Bonn, 1994.

[122] T. Zhang and C.-C. J. Kuo, "Hierarchical System for Content-based Audio Classification and Retrieval", *SPIE's Conference on Multimedia Storage and Archiving Systems III*, Boston, Nov. 1998.

[123] Steinberg Cubase
`http://www.steinberg.de`

[124] MET++
`www.ifi.unizh.ch/groups/mml/projects/met++/met++.html`

Die Gültigkeit aller angegebenen URLs wurde zuletzt am 26.2.2005 überprüft.

Erklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und unter ausschließlicher Verwendung der angegebenen Literatur angefertigt zu haben.